2

WRDC-TR-90-5021

AD-A230 396

A GENERALIZED EXTRACTION SYSTEM FOR VLSI

MICHAEL ALAN DUKES, M.S.E.E.
CAPTAIN, U.S. ARMY
AIR FORCE INSTITUTE OF TECHNOLOGY

FRANK MARKHAM BROWN, PhD
PROFESSOR OF ELECTRICAL ENGINEERING
AIR FORCE INSTITUTE OF TECHNOLOGY

JOANNE E. DEGROAT, PhD
ASSISTANT PROFESSOR OF ELECTRICAL ENGINEERING
OHIO STATE UNIVERSITY

October 1990

FINAL REPORT FOR PERIOD JUL 87 TO AUG 90

ELECTRONIC TECHNOLOGY LABORATORY
WRIGHT RESEARCH AND DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

NOTICE

This technical report has been reviewed and is approved for publication.

[illegible]
MICHAEL A. DUKES, CPT, USA
Air Force Institute of Technology

[illegible]
JOHN W. HINES, Chief
Design Branch
Microelectronics Division

FOR THE COMMANDER

[illegible]
STANLEY E. WAGNER, Chief
Microelectronics Division
Electronic Technology Laboratory

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE	*Form Approved OMB No 0704-0188*
1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) WRDC-TR-90-5021	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION AFT/ENG — 6b OFFICE SYMBOL *(If applicable)*	7a NAME OF MONITORING ORGANIZATION WRDC/ELED
6c ADDRESS *(City, State, and ZIP Code)* AFIT/ENG Wright-Patterson AFB OH 45433-6543	7b ADDRESS *(City, State, and ZIP Code)* WRDC/ELED Wright-Patterson AFB OH 45433-6543
8a NAME OF FUNDING/SPONSORING ORGANIZATION — 8b OFFICE SYMBOL *(If applicable)*	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER IN-HOUSE
8c ADDRESS *(City, State, and ZIP Code)*	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO 62204F — PROJECT NO 6096 — TASK NO 40 — WORK UNIT ACCESSION NO 18

11 TITLE *(Include Security Classification)*

A Generalized Extraction System for VLSI

12 PERSONAL AUTHOR(S)
Dukes, Michael Alan, Brown, Frank Markham, DeGroat, Joanne E.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT *(Year, Month, Day)*	15 PAGE COUNT
FINAL	FROM 07/87 TO 08/90	1990 October	110

16 SUPPLEMENTARY NOTATION

The computer software contained herein are "harmless". Already in the public domain

17 COSATI CODES

FIELD	GROUP	SUB-GROUP
12	05	
12	05	

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*

Formal Hardware Verification - Logic Programming - VLSI

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

A Generalized Extraction System for VLSI (GES, pronounced guess) is described. GES, which is written in Prolog, performs logic extraction from transistor netlists, indentification of logic errors within and between components, and generation of VHDL. The input to GES consists of a transistor netlist using the format from extract in magic. Logic extraction has been performed, on transistor netlists extracted form design layouts in magic, up to the level of 32-bit adders and 32-bit registers. GES reports typical errors in the construction and interconnection of components. An error-report identifies the component and specifies its location within the magic layout, making it easy to locate the offending circuitry. GES also provides a hierarchical VHDL description of the layout-design that is verified to be free of a large class of design errors.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL CAPT MICHAEL A. DUKES	22b TELEPHONE *(Include Area Code)* (513) 255-8626 — 22c OFFICE SYMBOL WRDC/ELED

DD Form 1473, JUN 86 *Previous editions are obsolete* SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Table of Contents

List of Figures

List of Acronyms

Acronym	Explanation
1076	IEEE Std 1076-1987
CAD	Computer Aided Design
cif	Caltech Intermediate Format
DoD	Department of Defense
esim	Switch level simulator
GND	Ground or Zero Volts
HDL	Hardware Description Language
HOL	Higher Order Logic
ML	Meta Language
MOS	Metal-Oxide-Semiconductor
mextra	Translate mask level format to transistor netlist format
sim	Transistor netlist generated by mextra
Vdd	Supplied Voltage, High, or +5 Volts
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language, IEEE Std 1076-1987

I. Introduction

A system for performing extraction of VLSI circuits is described in this paper. The system, called a Generalized Extraction System (GES, pronounced guess), contains three parts. The first part performs hierarchical extraction from component netlists. The second part identifies design errors within component netlists. The third part generates VHDL code from component netlists and may be viewed as the reverse of synthesis; a hardware description language is generated from a component netlist.

The form of extraction performed by GES entails identifying higher level components constructed from existing lower level components. In its simplest form, the extraction process may be viewed as identifying digital logic gates formed from a transistor netlist. The next level might involve identification of half adders from a component netlist of digital logic gates. The extraction process in GES may be started at any level to produce a higher and more abstract level of representation.

Extraction is a three-step process performed iteratively on a component netlist. The first step is to find a group of related components based on the component types and interconnections between them. Next, the identified components are eliminated from the component netlist. Finally, a new higher-level component (constructed from the identified components) is added to the component netlist.

GES may also be viewed as an expert system. A set of rules describes how higher level components may be constructed from lower level components. A component netlist is then supplied to the system as facts. Once the rules and facts are loaded, the system executes the rules against the fact base, generating new facts, representing higher level components that are derived from the lower level components.

Like the extraction process, the error identification process in GES is also constructed from rules that describe component configurations; however, these component configurations are deemed to be errors. The error identification rules are used against the component netlist to identify components that involve design errors. A report listing those components is generated for the user from GES.

The VHDL generation portion of GES generates VHDL from the existing component netlist. The component netlist may either be the original transistor netlist, a netlist of extracted components, or a combination netlist of transistors and logical components. A structural VHDL description and a HOL description are produced.

GES is written in Prolog and has been tested using Quintus Prolog[1] (under Ultrix[2], SunOS[3], and VAX/VMS[4]), 8086 Prolog-1[5], and C-Prolog[6](under UNIX[7]). GES was originally developed to extract VLSI custom layout designs generated in *magic* using the Berkeley Distribution of Design Tools [1]. Transistor netlists were generated from *magic* using either *extract* or *cif*[8]. Netlists from other computer aided design (CAD) tools may be input into GES after processing by an input filter which produces the netlist format required by GES. The users of this system may find other applications.

[1]Quintus and Quintus Prolog are trademarks of Quintus Computer Systems, Inc.
[2]Ultrix is a trademark of Digital Equipment Corporation
[3]SunOS is a trademark of Sun Microsystems, Incorporated
[4]VAX and VMS are trademarks of Digital Equipment Corporation
[5]Prolog86 for IBM PC-DOS
[6]C-Prolog was developed at EdCAAD, Department of Architecture, University of Edinburgh
[7]UNIX is a trademark of Bell Laboratories
[8]CALTECH Intermediate Format (CIF)

1.1 Assumptions

Some of the assumptions on which the presentation of this paper is based are listed below.

1. The user is familiar with "Clocksin and Mellish" Prolog as described in [2].
2. The user is familiar with VHDL ANSI/IEEE Std 1076-1987, the *IEEE Standard VHDL Language Reference Manual*[3].
3. The labeling convention used in the component netlist conforms to the identifier convention in §13.3 of the *IEEE Standard VHDL Language Reference Manual*[3][9] with the additional restriction that only upper case letters are used.

If the user is unfamiliar with Prolog, sufficient knowledge of Prolog for using GES may be gained by reading the first 30 pages of [4]. The second assumption is necessary if GES is being used to generate VHDL. The third assumption is necessary since VHDL not only limits the possible characters for an identifier, but is also case insensitive. Prolog, CIF, *magic*, and *mextra* distinguish different node identifiers by the upper or lower case of letters that exist in the node identifier.

1.2 Scope

This version of GES was developed primarily for extracting higher-level components from CMOS transistor netlists. However, GES is not limited to CMOS designs. Logic extraction from other components, e.g., resistors, bipolar transistors, and other technologies, may also be performed. If standard cell libraries can be represented as component netlists, then this tool can be used.

1.3 Limitations

Experience with GES has shown the extraction process to be ideal for parallel implementation. If the components in the component netlist appear relatively grouped according to their interconnectivity, a parallel implementation can reduce the number of "failed"[10] searches. Eliminating redundant components preempts parallelization; however, the process may be parallelized after eliminating redundant components. If internal error identification is being performed, the extraction process may not be parallelized.

1.4 Sequence of Presentation

This paper consists of several sections and appendices. The first section is an introduction to the paper as well as a list of assumptions and requirements for using GES. The second section includes a very general explanation for generating simple extraction rules. The third section outlines the format of the component netlist accepted by the system. The fourth section contains instructions on how to form the Prolog rules for extraction. The fifth section contains material on how to form Prolog rules for identifying design errors. The sixth section is a review of sections four and five. Contained in the review is a GES system that may be used to extract CMOS designs. The seventh section outlines the methodology used to generate VHDL code from component netlists. The eighth section presents two examples using GES. The ninth section contains some conclusions about the work. Finally, several appendices have been included; these define terms used in the paper and provide helpful code.

[9]identifier ::= letter {[underline] letter_or_digit }

[10]A failed search occurs when a component is pulled from the netlist for comparison but does not match the given extraction rule. Using a parallel implementation assumes that that components in the netlist are closely grouped. Comparing fewer components means fewer failures.

II. A Simple View of GES

The purpose of this section is to provide a simple overview of GES. The explanation that follows treats GES as a database system. Further, the discussion is oriented to CMOS transistor netlist extraction. It is hoped that this discussion will provide a user unfamiliar with Prolog an appreciation for its relational qualities.

Consider a format for a MOS transistor netlist

```
p(G,D,S,X,Y).
n(G,D,S,X,Y).
```

where p designates a p-type MOS transistor and n designates an n-type MOS transistor. Further, G denotes the gate, D the drain, S the source, X the X location, and Y the Y location according to *magic*. Uppercase first letters indicate variables and lowercase first letters indicate constants[1]. Thus, "vdd" or "clk_inv" would indicate constants and "In" or "PHI_PAR" would indicate variables. Now we have established the format representation in Prolog for the transistor netlist. An example CMOS netlist for two inverters, with the output of the first as the input to the second, would look like

```
p(in1,vdd,out1,50,50).
n(in1,gnd,out1,50,40).
p(out1,vdd,out2,100,50).
n(out1,gnd,out2,100,40).
```

The netlist, when loaded into Prolog, is called the fact list or database. In addition to the fact list, there are rules.

Rules in Prolog have the format *head* :- *body*. The *head* may include a list of arguments. The *body* is a set of conditions to be satisfied in order for the *head* to be true. To help simplify the extraction rules given later in the discussion, we will state some rules representing the interchangeability of the drain and source of a MOS transistor in a VLSI layout[2].

```
ptrans(G,D,S,X,Y) :-
  p(G,D,S,X,Y).
ptrans(G,D,S,X,Y) :-
  p(G,S,D,X,Y).
```

The above rules for **ptrans** have the following meaning: "A **ptrans** with arguments G,D,S,X,Y is defined by the condition that p has arguments G,D,S,X,Y **AND** a **ptrans** with arguments G,D,S,X,Y is defined by the condition that p has arguments G,S,D,X,Y." The arguments are position sensitive. Notice that S and D exchange positions in the bodies of the first and second rule. In the physical sense, the actual drain and source are determined by the biasing of the device. In the abstract sense (i.e., *magic* and *extract*), the drain and source are freely interchanged. The following shows a similar treatment for n-type transistors.

[1] In Prolog, a *constant* is a data object [4]. For this discussion, a *constant* would mean the actual node name.

[2] The physical aspect of the source and drain may differ from the abstract representation of the netlist. This difference is only due to the physical bias of the transistor determined by the flow of current.

```
ntrans(G,D,S,X,Y) :-
  n(G,D,S,X,Y).
ntrans(G,D,S,X,Y) :-
  n(G,S,D,X,Y).
```

Assuming an inverter can be found, we will eliminate the transistors that construct the inverter and then add to the database a fact representing the new-found inverter. The following rule performs the stated tasks.

```
inv :-
  ptrans(In,vdd,Out,X1,Y1),
  ntrans(In,gnd,Out,_,_),
  remove_p(In,vdd,Out),
  remove_n(In,gnd,Out),
  asserta(inv(In,Out,X1,Y1)),
  fail.
inv.
```

The above rule for interacting with the database may be interpreted in the following manner. "**inv** is defined by a **ptrans** component with arguments In,vdd,Out,X1,Y1 and an **ntrans** component with arguments In,gnd,Out,_,_." The "_" denotes a "don't care" for the value assumed by that field. This means that the first two conditions,

```
ptrans(In,vdd,Out,X1,Y1),
ntrans(In,gnd,Out,_,_),
```

must be satisfied before we can start removing transistors and adding an inverter in their place in the database. The removal of the transistors is accomplished in the following manner.

```
remove_p(G,D,S) :-
  retract(p(G,D,S,_,_)).
remove_p(G,D,S) :-
  retract(p(G,S,D,_,_)).

remove_n(G,D,S) :-
  retract(n(G,D,S,_,_)).
remove_n(G,D,S) :-
  retract(n(G,S,D,_,_)).
```

Like the **ptrans** and **ntrans** rules, rules for allowing the source and drain to be switched must be taken into account. Within the above rules, **retract** is a Prolog function for specifically removing facts from the database. The Prolog function **asserta** is used to add facts to the Prolog database. The Prolog function fail is used to force the Prolog rule inv to backup and find another pair of transistors that satisfy the conditions of the rule. Without the fail Prolog function, the inv Prolog rule would only attempt to extract one transistor.

The UNIX script file listing below illustrates the operation of GES. Prolog is invoked at the system prompt by entering prolog. The file containing the transistor netlist is called db. The file containing the rules for querying the database is called w rk. Prolog prompts the user for input using '| ?-' which is the system's way of asking for a database query to be satisfied. First, the files db and work are loaded using ['db'] and [ork']. After the files are loaded, the database is examined using the Prolog function listing. The query inv is then typed at the Prolog prompt. The listing function is then used again to see that the transistors have been replaced by inverters. The system is exited by typing halt.

Below is a listing of the transistor netlist in db.

```
Script started on Sat Jun 16 00:40:00 1990
csh> more db
    p(in1,vdd,out1,50,50).
    p(out1,vdd,out2,100,50).
    n(in1,gnd,out1,50,40).
    n(out1,gnd,out2,100,40).
```

A listing of the Prolog extraction rules is shown below.

```
csh> more work
    ptrans(G,D,S,X,Y) :-
      p(G,D,S,X,Y).
    ptrans(G,D,S,X,Y) :-
      p(G,S,D,X,Y).
    ntrans(G,D,S,X,Y) :-
      n(G,D,S,X,Y).
    ntrans(G,D,S,X,Y) :-
      n(G,S,D,X,Y).

    inv :-
      ptrans(In,vdd,Out,X1,Y1),
      ntrans(In,gnd,Out,_,_),
      remove_p(In,vdd,Out),
      remove_n(In,gnd,Out),
      asserta(inv(In,Out,X1,Y1)),
      fail.
    inv.

    remove_p(G,D,S) :-
      retract(p(G,D,S,_,_)).
    remove_p(G,D,S) :-
      retract(p(G,S,D,_,_)).
    remove_n(G,D,S) :-
      retract(n(G,D,S,_,_)).
    remove_n(G,D,S) :-
      retract(n(G,S,D,_,_)).
```

Invoking Prolog in UNIX is shown below.

```
csh> prolog

Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?-
```

Loading the transistor netlist and the Prolog extraction rules into Prolog is shown below.

```
| ?- ['db'].
[consulting /usr/users/eled/dukes/admin/db...]
[db consulted 0.133 sec 792 bytes]

yes
| ?- ['work'].
[consulting /usr/users/eled/dukes/admin/work...]
[work consulted 0.333 sec 1,276 bytes]

yes
```

The method for displaying the contents of the Prolog database is shown below.

```
| ?- listing.

library_directory('/usr/local/q2.4/library').
library_directory('/usr/local/q2.4/tools').
library_directory('/usr/local/q2.4/IPC').

p(in1,vdd,out1,50,50).
p(out1,vdd,out2,100,50).

n(in1,gnd,out1,50,40).
n(out1,gnd,out2,100,40).

ptrans(A,B,C,D,E) :-
        p(A,B,C,D,E).
ptrans(A,B,C,D,E) :-
        p(A,C,B,D,E).
```

```
ntrans(A,B,C,D,E) :-
        n(A,B,C,D,E).
ntrans(A,B,C,D,E) :-
        n(A,C,B,D,E).

inv :-
        ptrans(A,vdd,B,C,D),
        ntrans(A,gnd,B,E,F),
        remove_p(A,vdd,B),
        remove_n(A,gnd,B),
        asserta(user:inv(A,B,C,D)),
        fail.
inv.

remove_p(A,B,C) :-
        retract(user:p(A,B,C,D,E)).
remove_p(A,B,C) :-
        retract(user:p(A,C,B,D,E)).

remove_n(A,B,C) :-
        retract(user:n(A,B,C,D,E)).
remove_n(A,B,C) :-
        retract(user:n(A,C,B,D,E)).

yes
```

Invoking the Prolog extraction rule inv is shown below, as well as, a listing of the contents of the Prolog database after extraction.

```
| ?- inv.

yes
| ?- listing.

library_directory('/usr/local/q2.4/library').
library_directory('/usr/local/q2.4/tools').
library_directory('/usr/local/q2.4/IPC').

ptrans(A,B,C,D,E) :-
        p(A,B,C,D,E).
ptrans(A,B,C,D,E) :-
        p(A,C,B,D,E).

ntrans(A,B,C,D,E) :-
        n(A,B,C,D,E).
ntrans(A,B,C,D,E) :-
        n(A,C,B,D,E).
```

```
inv(out1,out2,100,50).
inv(in1,out1,50,50).

inv :-
        ptrans(A,vdd,B,C,D),
        ntrans(A,gnd,B,E,F),
        remove_p(A,vdd,B),
        remove_n(A,gnd,B),
        asserta(user:inv(A,B,C,D)),
        fail.
inv.

remove_p(A,B,C) :-
        retract(user:p(A,B,C,D,E)).
remove_p(A,B,C) :-
        retract(user:p(A,C,B,D,E)).

remove_n(A,B,C) :-
        retract(user:n(A,B,C,D,E)).
remove_n(A,B,C) :-
        retract(user:n(A,C,B,D,E)).

yes
| ?- halt.
csh>
script done on Sat Jun 16 00:40:34 1990
```

We can see above that the four original MOS transistors in the transistor netlist have been replaced by two inverters.

III. Component Netlist Format for GES

This section describes the format for GES input. Also presented are some routines for converting one transistor format into the format required for GES.

3.1 Generating Transistor Netlists from Magic

The required form for a transistor netlist is that described in [5]. This form was chosen since it was derived from the mask layout form of *magic* used in [1]. One process of generating the transistor netlist begins in *magic* by using the :cif command. The :cif command in *magic* produces a mask layout file in CIF. Afterwards, *mextra* reads in the CIF file and produces a transistor netlist file. The record format for a transistor is

```
type gate source drain length width xpos ypos
```

where **type** is one of e, p, or n for enhancement mode transistor, p-type transistor, or n-type transistor, respectively. The second through fourth fields, **gate**, **source**, and **drain**, describe three of the four terminals of the MOS transistor used. The base is assumed to be biased correctly and is not included. The fifth and sixth fields, **length** and **width**, describe the channel length and channel width of the MOS transistor. The seventh and eighth fields, **xpos** and **ypos**, are the location coordinates of the transistor.

An alternate route for extracting a transistor netlist from a *magic* layout also exists. The :extract command in *magic* creates a hierarchical form of the layout, in **.ext** format, currently residing in *magic*. A series of extraction files is created that reflects the cell hierarchy used in *magic*. A tool called *ext2sim* is then used to generate the transistor netlist form.

3.2 Converting Transistor Netlists to Prolog Clause Form

If the *mextra* tool is used on a CIF file, the transistor netlist will be created with the n-type and p-type transistors described as e and p for their types, respectively. If the *ext2sim* tool is used, the transistor netlist will be created with n-type and p-type transistors described as n and p for their types, respectively. Thus, a transistor generated from *mextra* as

```
e a_XNOR_c#17 1520 GND 300 1200 706200 -20550
```

would appear in Prolog clause form as

$$n(nA\_XNOR\_C17, n1520, ngnd, 300, 1200, 706200, -20550).$$

A transistor generated from *ext2sim* as

```
p 20/4_1/A_in_nand Vdd 20/4_1/probe 300 1200 12172 101
```

would appear in Prolog clause form as

$$p(n204\_1A\_IN\_NAND, nvdd, n204\_1PROBE, 300, 1200, 12172, 101).$$

The general transistor netlist format in Prolog is shown below.

$$type_P(gate_P, source_P, drain_P, xpos, ypos).$$

The following mappings are used:

$$\begin{array}{rcl}
type & \mapsto & type_P \\
p & \mapsto & p \\
n & \mapsto & n \\
e & \mapsto & n \\
 & & \\
gate & \mapsto & gate_P \\
node & \mapsto & nNODE \\
Vdd & \mapsto & nvdd \\
GND & \mapsto & ngnd \\
Gnd & \mapsto & ngnd \\
 & & \\
source & \mapsto & source_P \\
node & \mapsto & nNODE \\
Vdd & \mapsto & nvdd \\
GND & \mapsto & ngnd \\
Gnd & \mapsto & ngnd \\
 & & \\
drain & \mapsto & drain_P \\
node & \mapsto & nNODE \\
Vdd & \mapsto & nvdd \\
GND & \mapsto & ngnd \\
Gnd & \mapsto & ngnd
\end{array}$$

where

$$node = \{numbers, lowercase, uppercase, specialcharacters\}$$

and

$$nNODE = n * \{numbers, uppercase\}.$$

The * is used as the concatenate operation. The set $specialcharacters = \{@, \#, \%, *, (,), /, [,], ', ", `\}$ is not necessarily a complete one since *magic* allows labels to consist of a large number of different special symbols. The process $node \mapsto nNODE$ drops *specialcharacters* and translates *lowercase* to *uppercase*.

A copy of a `lex` program for mapping node labels to uppercase is shown in Appendix B. A copy of a C program that converts a transistor netlist from *mextra* to Prolog form is shown in Appendix C. The `lex` program is run before *mextra* since *mextra* renames labels that are equivalent but on different nodes.

IV. Writing Prolog Extraction Rules

This section presents general guidelines for writing Prolog extraction rules. Examples of Prolog extraction rules are included. The extraction rules listed here assume CMOS design. Rules for other technologies may be developed using these rules as a guideline.

4.1 Levels of Rules

Extraction from lower-level components to higher-level components may be performed in one of two ways. One method is to extract the highest level component through recognition of all of the lowest level components that compose it; however, this method is impractical and only mentioned for completeness. A second possible method is to extract components recognizing intermediate levels of components in a hierarchical fashion. The second method is explained in this section. Afterwards, a discussion of the problems using the first method will be presented.

4.2 Ground-Rule Set and Fact Set

The ground-rule set and fact set elaborate the initial conditions of the logic extraction process. The ground-rule set describes the basic p-type and n-type MOS transistor terminals.

Definition 1 $p(G,D,S,W,L,X,Y)$ is a predicate that describes a p-type MOS transistor with a gate G, drain D, source S, channel width W, channel length L, x-location X, and y-location Y.

Definition 2 $n(G,D,S,W,L,X,Y)$ is a predicate that describes an n-type MOS transistor with a gate G, drain D, source S, channel width W, channel length L, x-location X, and y-location Y.

The arity (number of arguments) of seven for both the p and n predicates assumes that the physical base connection of the MOS transistor is biased correctly. Furthermore, the description adopted is for p-type and n-type enhancement mode MOS transistors. Typical p-type and n-type MOS transistors in Prolog appear as

```
p(nINPUT,nvdd,nOUTPUT,3,6,1254,387).
p(nADDIN,nA_INPUT,nA_SELECT,3,6,39887,-3091).
n(nINPUT,ngnd,nOUTPUT,3,6,1260,387).
```

In the physical sense, the actual drain and source are determined by the biasing of the device. In the abstract sense (i.e., *magic* and *extract*), the drain and source are freely interchanged. Implementations of the p-type and n-type transistors in MOS layout freely interchange the drain and source [6]. In order to express this abstract aspect and to suppress information concerning length and width, some further definitions are adopted.

Definition 3 The predicate 'ptrans' is defined in terms of the predicate 'p' by the implication

$$\forall G,D,S,X,Y\ [(p(G,D,S,\_,\_,X,Y) \vee p(G,S,D,\_,\_,X,Y)) \Rightarrow ptrans(G,D,S,X,Y)].$$

Definition 4 The predicate 'ntrans' is defined in terms of the predicate 'n' by the implication

$$\forall G,D,S,X,Y\ [(n(G,D,S,\_,\_,X,Y) \vee n(G,S,D,\_,\_,X,Y)) \Rightarrow ntrans(G,D,S,X,Y)].$$

The underscore indicates unnecessary information. The Prolog rules to describe Definitions 3 and 4 are

```
ptrans(G,D,S,X,Y) :-
  p(G,D,S,_,_,X,Y).
ptrans(G,D,S,X,Y) :-
  p(G,S,D,_,_,X,Y).

ntrans(G,D,S,X,Y) :-
  n(G,D,S,_,_,X,Y).
ntrans(G,D,S,X,Y) :-
  n(G,S,D,_,_,X,Y).
```

For clarity and brevity, we use the notation *predicate/arity*. Thus, **ptrans/5** and **ntrans/5** represent the above two Prolog rules, respectively.

Assume that **ptrans/5** and **ntrans/5** exist in a file called **trans.pro** on a UNIX system and that Quintus Prolog is also installed on the same system. **ptrans/5** and **ntrans/5** may be loaded into Prolog in the following manner.

```
% prolog

Quintus Prolog Release 2.2 (Sun-3, Unix 3.2)
Copyright (C) 1987, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- compile(['trans.pro']).
[compiling /people/dukes/class/trans.pro...]
[trans.pro compiled 0.267 sec 564 bytes]

yes
| ?-
```

The system prompt is **%**. The Prolog prompt is **| ?-** . The file, **trans.pro**, was loaded into Prolog using the Quintus Prolog procedure called **compile/1**. The Prolog function, **compile/1**, compiles the contents of the file, **trans.pro**, into the current Prolog session.

Assume the following transistor netlist exists in a UNIX file called intrans.pro.

```
p(nINPUT,nvdd,nOUTPUT,3,6,1254,387).
p(nADDIN,nA_INPUT,nA_SELECT,3,6,39887,-3091).
n(nINPUT,ngnd,nOUTPUT,3,6,1260,387).
```

The transistor information would be read into Prolog in the following manner.

```
| ?- ['intrans.pro'].
[consulting /people/dukes/class/intrans.pro...]
[intrans.pro consulted 0.100 sec 564 bytes]

yes
| ?-
```

All of the p-type transistors may be listed by querying Prolog in the following manner.

```
| ?- p(G,D,S,W,L,X,Y).

G = nINPUT,
D = nvdd,
S = nOUTPUT,
W = 3,
L = 6,
X = 1254,
Y = 387 ;

G = nADDIN,
D = nA_INPUT,
S = nA_SELECT,
W = 3,
L = 6,
X = 39887,
Y = -3091 ;

no
| ?- halt.

[ End of Prolog execution ]
%
```

The upper case letters, G, D, S, W, L, X, and Y, designate variables to be instantiated by Prolog. The ; (semicolon) is used to request further information from the transistor database that satisfied the request. Otherwise, a carriage return not preceded by a ; would have terminated the search. The `halt/0` predicate tells Prolog to terminate and return to the system prompt.

Assume now that the transistor netlist consists of the following components.

```
p(nINPUT,nvdd,nOUTPUT,3,6,1254,387).
p(nINSTATE,nNOTINSTATE,nvdd,3,6,1254,387).
p(nADDIN,nA_INPUT,nA_SELECT,3,6,39887,-3091).
n(nINPUT,ngnd,nOUTPUT,3,6,1260,387).
```

Assume also that we are interested in finding transistors with nvdd on either the drain or source of a p-type transistor. The objective may be accomplished in one of two ways. The first method is to express two queries to Prolog using

```
| ?- p(G,nvdd,S,W,L,X,Y) ; p(G,D,nvdd,W,L,X,Y).
```

In the above example, the ; is used to logically `OR` the two queries. The result of the two queries is displayed below.

```
% prolog

Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
```

```
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- ['intrans.pro'].
[consulting /people/dukes/class/intrans.pro...]
[intrans.pro consulted 0.167 sec 880 bytes]

yes
| ?- p(G,nvdd,S,W,L,X,Y) ; p(G,D,nvdd,W,L,X,Y).

G = nINPUT,
S = nOUTPUT,
W = 3,
L = 6,
X = 1254,
Y = 387,
D = _149 ;

G = nINSTATE,
S = _55,
W = 3,
L = 6,
X = 1254,
Y = 387,
D = nNOTINSTATE ;

no
| ?-
```

However, the Prolog rules stated in ptrans/5 will accomplish the same task, as shown in the following:

```
% prolog

Quintus Prolog Release 2.2 (Sun-3, Unix 3.2)
Copyright (C) 1987, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- compile(['trans.pro']).
[compiling /people/dukes/class/trans.pro...]
[trans.pro compiled 0.250 sec 564 bytes]

yes
| ?- ['intrans.pro'].
[consulting /people/dukes/class/intrans.pro...]
[intrans.pro consulted 0.100 sec 688 bytes]

yes
| ?- ptrans(G,nvdd,S,X,Y).

G = input,
S = output,
X = 1254,
Y = 387 ;

G = instate,
S = notinstate,
X = 1254,
Y = 387 ;

no
| ?-
```

This example demonstrates that rules using transistors may not be concerned with the interchangeability of the drain and source. Assuming that circuit-function, not timing, is of primary interest, unnecessary information (e.g., gate width and length) may be easily dropped when performing extraction.

4.3 Level-1 Prolog Rules

The level-1 Prolog rules describe basic components, i.e., inverters, transmission gates, and high-impedance inverters. These rules are derived from definitions 5, 6, and 8 respectively, which are given below. Also, definition 7 describes the bidirectionality of the input and output of transmission gates. The level-2 Prolog rules, discussed in the succeeding section, describe logic gates constructed completely from transistors, i.e., NAND gates and NOR gates. Finally, level-N Prolog rules, described after level-2 Prolog rules, are for components that are constructed from a mixture of components of level-1, level-2, transistors, and higher level Prolog rules.

The purpose of providing the previous definitions and the definitions that follow is to demonstrate that the extraction rules are based on logic. The extraction process is a method of proving the existence of higher level constructs from existing lower-level ones. By seeing a relation between definitions and rules, we may demonstrate that the extraction process is a form of formal verification. In fact, the highest level Prolog rule may also be the final goal to be achieved in an extraction process whereby only one component is left over after the entire extraction process has run its course.

For notational convenience and clarity the following symbols will be used. Let I denote an input node, O denote an output node, IO denote a bidirectional node, *nvdd* represent the circuit voltage for Vdd, *ngnd* represent the circuit voltage for GND, P and N represent additional outside stimuli for the circuit under consideration, Q and R represent internal circuit wiring, X represent the X location, and Y represent the Y location.

Definition 5 The predicate 'inv' is defined in terms of the predicates 'ptrans' and 'ntrans' by the implication

$$\forall I, O, X, Y\ [(ptrans(I, nvdd, O, X, Y) \wedge ntrans(I, ngnd, O, \_, \_)) \Rightarrow inv(I, O, \_, \_)].$$

Definition 6 The predicate 'tgate' is defined in terms of the predicates 'ptrans' and 'ntrans' by the implication

$$\forall P, N, IO_1, IO_2, X, Y \ [(ptrans(P, IO_1, IO_2, X, Y) \wedge ntrans(N, IO_1, IO_2, \_, \_)) \Rightarrow tgate(P, N, IO_1, IO_2, \_, \_)].$$

Definition 7 The predicate 'tgate_rule' is defined in terms of the predicate 'tgate' by the implication

$$\forall P, N, IO_1, IO_2, X, Y \ [(tgate(P, N, IO_1, IO_2, X, Y) \vee tgate(P, N, IO_2, IO_1, X, Y) \Rightarrow tgate\_rule(P, N, IO_1, IO_2, X, Y))]$$

Definition 8 The predicate 'invZ' is defined in terms of the predicates 'ptrans' and 'ntrans' by the implication

$$\forall P, N, I, O, Q, R, X, Y \ [(ptrans(I, nvdd, Q, X, Y) \wedge ptrans(P, Q, O, \_, \_) \wedge ntrans(N, O, R, \_, \_) \wedge ntrans(I, R, ngnd, \_, \_) \wedge (Q \neq R)) \Rightarrow invZ(P, N, I, O, X, Y)].$$

Definitions 5, 6, and 8 are described in Prolog below.

```
inv(I,O,X,Y) :-
  ptrans(I,nvdd,O,X,Y),
  ntrans(I,ngnd,O,_,_).

tgate(P,N,IO1,IO2,X,Y) :-
  ptrans(P,IO1,IO2,X,Y),
  ntrans(N,IO1,IO2,_,_).

invZ(P,N,I,O,X,Y) :-
  ptrans(I,nvdd,Q,X,Y),
  ptrans(P,Q,O,_,_),
  ntrans(N,O,R,_,_),
  ntrans(I,R,ngnd,_,_),
  Q \== R.
```

In each case above, the X and Y location was chosen from the first occurring p-type transistor. We might have chosen to calculate the X and Y location of the resulting component from computing the average X and Y values from all of the lower-level components. An example of this would be the following.

```
invZ(P,N,I,O,X,Y) :-
  ptrans(I,nvdd,Q,X1,Y1),
  ptrans(P,Q,O,X2,Y2),
ntrans(N,O,R,X3,Y3),
ntrans(I,R,ngnd,X4,Y4),
Q \== R,
X is (X1 + X2 + X3 + X4)/4,
Y is (Y1 + Y2 + Y3 + Y4)/4.
```

The condition `Q \== R` is part of the Prolog rule `invZ/6`. Providing different variable names within a Prolog clause does not prevent them from assuming the same value. If the two variables representing two signal lines must have different values, it is imperative that this condition be placed within the rule after the variables have been declared.

The Prolog rules stated for `inv/4`, `tgate/6`, and `invZ/6` may also be placed in `trans.pro` with `ptrans/5` and `ntrans/5`. To demonstrate the usefulness of these Prolog rules, the transistor netlist from the previous section will be queried for the existence of inverters, transmission gates, and high-impedance inverters. The previous transistor netlist is repeated below.

```
p(nINPUT,nvdd,nOUTPUT,3,6,1254,387).
p(nINSTATE,nNOTINSTATE,nvdd,3,6,1254,387).
p(nADDIN,nA_INPUT,nA_SELECT,3,6,39887,-3091).
n(nINPUT,ngnd,nOUTPUT,3,6,1260,387).
```

The file containing the transistor netlist is called `intrans.pro`. The following is the Prolog session.

```
% prolog

Quintus Prolog Release 2.2 (Sun-3, Unix 3.2)
Copyright (C) 1987, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- compile(['trans.pro']).
[compiling /people/dukes/class/trans.pro...]
[trans.pro compiled 0.534 sec 952 bytes]

yes
| ?- ['intrans.pro'].
[consulting /people/dukes/class/intrans.pro...]
[intrans.pro consulted 0.067 sec 680 bytes]

yes
| ?- inv(In,Out,X,Y).

In = nINPUT,
Out = nOUTPUT,
X = 1254,
Y = 387 ;

no
| ?- tgate(P,N,In,Out,X,Y).

no
| ?- invZ(P,N,In,Out,X,Y).
```

```
no
| ?-
```

From the transistor netlist in **intrans.pro** there was one inverter. No transmission gates or high-impedance inverters were found. Inspection of the transistor netlist would indicate that there is only one inverter present.

Up to this point, the discussion has focused on the declarative nature of the extraction problem. In order to conduct the extraction process in an automated fashion, some procedural aspects of Prolog must be considered.

When a component is extracted from lower level components, information regarding the lower level components is no longer necessary. However, the component built up from the lower level components must be declared for later use. Accordingly, five procedural steps are performed in the order indicated.

1. Identify the component from its lower-level components.
2. Check that the values of internal nodes do not match other nodes.
3. Delete the lower-level components from the component netlist.
4. Add the newly found component to the component netlist.
5. Check to see if there are more lower-level components.

Step 1 must occur first, step 2 must occur second, and step 5 must occur last; however, the order of steps 3 and 4 is not important. To illustrate how the above steps are incorporated into a Prolog rule, a Prolog extraction rule will be formed from **invZ/6**.

From **invZ/6**, there are four subcomponents that must be identified.

```
  ptrans(I,nvdd,Q,X,Y)
  ptrans(P,Q,O,_,_)
  ntrans(N,O,R,_,_)
  ntrans(I,R,ngnd,_,_)
```

There are two nodes, Q and R, that are required to be separate from the rest of the nodes. To guarantee this condition, the following definition and Prolog rules are offered.

Definition 9 The predicate 'not_connected' is defined by the equivalence

$$(not\_connected([Node_1, ..., Node_n]) = true) \Leftrightarrow (\forall i, j \in n \ Node_i \neq Node_j)$$

where n is the number of nodes.

The Prolog rules to implement Definition 9 are:

```
not_connected([]) :- !.
not_connected([Node|Tail]) :-
  not_member(Node,Tail),
  not_connected(Tail).

not_member(_,[]) :- !.
not_member(Node,[Head|Tail]) :-
  Node \== Head,
  not_member(Node,Tail).
```

In order to use this new Prolog rule, a list of nodes is provided to **not_connected/1**. For **invZ/6**, the following Prolog clause would be added.

```
  not_connected([Q,R,ngnd,nvdd])
```

The not_connected/1 rule is used to ensure that Q, R, ngnd, and nvdd are not connected. Though it seems obvious that nvdd and ngnd should not be connected, they are included to check that neither Q nor R is connected to nvdd or ngnd.

Within Prolog are two procedures for manipulating a facts database. The procedures are **retract/1** and **assert/1**. The **retract/1** procedure removes an occurrence of a clause from the facts database. The **assert/1** predicate puts a clause into the database of clauses. The relative position of the new clause with respect to other clauses is nondeterministic. A variant of **assert/1**, called **asserta/1**, is used to create a new clause at the head of the database. Normally, **retract/1** is used explicitly; however, the problem of the interchangeability of the drain and source of the MOS transistor prevents direct use of **retract/1** when removing a clause **p/7** or a clause **n/7**. Therefore, some Prolog rules will be declared to allow for removal of the p-type and n-type transistors, keeping in mind the interchangeability of the drain and the source.

```
remove_p(G,D,S) :-
  retract(p(G,D,S,_,_,_,_)),!.
remove_p(G,D,S) :-
  retract(p(G,S,D,_,_,_,_)),!.

remove_n(G,D,S) :-
  retract(n(G,D,S,_,_,_,_)),!.
remove_n(G,D,S) :-
  retract(n(G,D,S,_,_,_,_)),!.
```

With the new Prolog rules **remove_p/3** and **remove_n/3**, we may delete the identified lower-level components. Continuing the **invZ/6** example, we would employ the goals

```
  remove_p(I,nvdd,Q)
  remove_p(P,Q,O)
  remove_n(N,O,R)
  remove_n(I,R,ngnd)
```

to remove the transistors that form **invZ/6**. The next step involves adding a new component to the component netlist. The following goal adds the new **invZ/6** component.

```
  asserta(invZ(P,N,I,O,X,Y))
```

The method used to continue finding components in Prolog is performed by placing a **fail/0** procedure at the end of a Prolog rule. The **invZ/0** Prolog rule is completed by adding **invZ.** after **fail/0** so that **invZ/0** may succeed as a goal. Thus, the entire Prolog rule for extracting all high-impedance inverters automatically would appear as follows.

```
invZ :-
  ptrans(I,nvdd,Q,X,Y),
  ptrans(P,Q,O,_,_),
  ntrans(N,O,R,_,_),
  ntrans(I,R,ngnd,_,_),
  not_connected([Q,R,ngnd,nvdd]),
  remove_p(I,nvdd,Q),
  remove_p(P,Q,O),
  remove_n(N,O,R),
  remove_n(I,R,ngnd),
  asserta(invZ(P,N,I,O,X,Y)),
  fail.
invZ.
```

The general template for forming Prolog extraction rules is:

```
head :-
  matching_goal_1,
        .
        .
        .
  matching_goal_n,
  not_connected([ internal_argument_list ]),
  retract_goal_1,
        .
        .
        .
  retract_goal_n,
  asserta(head(argument_list)),
  fail.
head.
```

In keeping with the five procedural steps mentioned earlier, the *matching_goals* correspond to step 1, `not_connected/1` corresponds to step 2, the *retract_goals* correspond to step 3, `assert/1` corresponds to step 4, and `fail/0` corresponds to step 5.

In order to be able retract facts loaded into Quintus Prolog, they need to be declared dynamic using the directive `dynamic/0`. To place the transistors in the dynamic storage area of Quintus, the line

```
:- dynamic p/7,n/7.
```

must be added to the beginning of `intrans.pro`.

To display the new rule formed for `invZ/0` we will use a new file called `intrans2.pro`. We will also replace `invZ/6` with `invZ/0` and add `remove_p/3` and `remove_n/3` to the file `trans.pro`.

```
% cat intrans.pro
:- dynamic p/7,n/7.
p(nGATED,nALPHA,nBETA,3,6,783,132).
p(nRIGHT_SELECT,n30_21_FILL,n30_21_FILLOUT,3,6,20985,23).
```

```
p(nINPUT,nvdd,nINTER1,3,6,1254,387).
p(nSELECT,nINTER1,nOUTPUT,3,6,1254,387).
n(nSELECTBAR,nINTER2,nOUTPUT,3,6,39887,-3091).
n(nINPUT,ngnd,nINTER2,3,6,1260,387).
% cat trans.pro
ptrans(G,D,S,X,Y) :-
  p(G,D,S,_,_,X,Y).
ptrans(G,D,S,X,Y) :-
  p(G,S,D,_,_,X,Y).

ntrans(G,D,S,X,Y) :-
  n(G,D,S,_,_,X,Y).
ntrans(G,D,S,X,Y) :-
  n(G,S,D,_,_,X,Y).

remove_p(G,D,S) :-
  retract(p(G,D,S,_,_,_,_)),!.
remove_p(G,D,S) :-
  retract(p(G,S,D,_,_,_,_)),!.

remove_n(G,D,S) :-
  retract(n(G,D,S,_,_,_,_)),!.
remove_n(G,D,S) :-
  retract(n(G,S,D,_,_,_,_)),!.

invZ :-
  ptrans(I,nvdd,Q,X,Y),
  ptrans(P,Q,O,_,_),
  ntrans(N,O,R,_,_),
  ntrans(I,R,ngnd,_,_),
  not_connected([Q,R,ngnd,nvdd]),
  remove_p(I,nvdd,Q),
  remove_p(P,Q,O),
  remove_n(N,O,R),
  remove_n(I,R,ngnd),
  asserta(invZ(P,N,I,O,X,Y)),
  fail.
invZ.

not_connected([]) :- !.
not_connected([Node|Tail]) :-
  not_member(Node,Tail),
  not_connected(Tail).

not_member(_,[]) :- !.
not_member(Node,[Head|Tail]) :-
  Node \== Head,
  not_member(Node,Tail).
% prolog

Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
```

```
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- compile(['trans.pro']).
[compiling /people/dukes/class/trans.pro...]
[trans.pro compiled 1.334 sec 1,424 bytes]

yes
| ?- ['intrans.pro'].
[consulting /people/dukes/class/intrans.pro...]
[intrans.pro consulted 0.167 sec 692 bytes]

yes
| ?-
```

After loading in **trans.pro** and **intrans.pro**, the Prolog extraction rule, **invZ/0**, will be used to identify any high-impedance inverters that exist in the database. Afterwards, the database will be examined. The commands to execute invZ/0 and examine the database are **invZ** and **listing** respectively.

```
| ?- invZ.

yes
| ?- listing.

library_directory('/usr/local/q2.4/library').
library_directory('/usr/local/q2.4/tools').
library_directory('/usr/local/q2.4/IPC').

p(nGATED,nALPHA,nBETA,3,6,783,132).
p(nRIGHT_SELECT,n30_21_FILL,n30_21_FILLOUT,3,6,20985,23).

invZ(nSELECT,nSELECTBAR,nINPUT,nOUTPUT,1254,387).

yes
| ?- halt.
```

The **listing/0** procedure in Quintus Prolog is used to examine the components present in the component database. Notice that the result of extracting the high-impedance inverter from the transistor netlist is the identification of one high-impedance inverter and two leftover p-type transistors.

Similar Prolog rules may be specified for identifying transmission gates and inverters. The Prolog rules are listed below.

```
tgate :-
  ptrans(P,IO1,IO2,X,Y),
  ntrans(N,IO1,IO2,_,_),
  remove_p(P,IO1,IO2),
  remove_n(N,IO1,IO2),
  asserta(tgate(P,N,IO1,IO2,X,Y)),
  fail.
tgate.

inv :-
  ptrans(I,O,nvdd,Xloc,Yloc),
  ntrans(I,ngnd,O,_,_),
  remove_p(I,O,nvdd),
  remove_n(I,ngnd,O),
  asserta(inv(I,O,Xloc,Yloc)),
  fail.
inv.
```

As with the p-type and n-type transistors, transmission gates require an additional rule for the interchangeability of the input/output ports. Using Definition 7, the Prolog rule for transmission gates is shown below.

```
tgate_rule(P,N,I,O,X,Y) :-
  tgate(P,N,I,O,X,Y).
tgate_rule(P,N,I,O,X,Y) :-
  tgate(P,N,O,I,X,Y).
```

For retracting transmission gates, a Prolog rule will also have to be created similar to **remove_p/3**. This Prolog rule is shown below.

```
remove_tgate(P,N,O,I) :-
  retract(tgate(P,N,O,I,_,_)),!.
remove_tgate(P,N,O,I) :-
  retract(tgate(P,N,I,O,_,_)).
```

4.4 Level-2 Prolog Rules

The Prolog rules discussed in this section describe those logic gates that are constructed from transistors without any other basic components described in the level-1 Prolog rules. The CMOS components that will be discussed in this section are the **NAND** gate and the **NOR** gate. Since the process of forming rules was covered in the previous section, there is no need for the level of detail presented earlier. The discussion that follows should help to reinforce the process of forming extraction rules.

The first component to be discussed is the **NAND** gate. Definition 10 is a description of how a two-input **NAND** gate is formed in CMOS.

Definition 10 The predicate 'nand' is defined in terms of the predicates 'ptrans', 'ntrans', and 'not_connected' by the implication

$$\begin{aligned}\forall A,B,O,Q,X,Y\ [(&ntrans(A,ngnd,Q,X,Y) \wedge ntrans(B,Q,O,\_,\_)\\ &\wedge ptrans(A,nvdd,O,\_,\_) \wedge ptrans(B,nvdd,O,\_,\_)\\ &\wedge not\_connected([Q,O,nvdd,ngnd]) \wedge not\_connected([Q,A])\\ &\wedge not\_connected([Q,B]))\\ &\qquad \Rightarrow \quad nand([A,B],O,X,Y)].\end{aligned}$$

The Prolog extraction rule for Definition 10 is:

```
nand :-
  ntrans(A,ngnd,Q,X,Y),
  ntrans(B,Q,O,_,_),
  ptrans(A,nvdd,O,_,_),
  ptrans(B,nvdd,O,_,_),
  not_connected([Q,O,nvdd,ngnd]),
  not_connected([Q,A]),
  not_connected([Q,B]),
  remove_n(A,ngnd,Q),
  remove_n(B,Q,O),
  remove_p(A,nvdd,O),
  remove_p(B,nvdd,O),
  asserta(nand([A,B],O,X,Y)),
  fail.
nand.
```

If we wish to find NAND gates with two or three inputs, additional extraction rules may be added to the one above. An example of a Prolog rule that extracts two-input and three-input NAND gates is shown below.

```
nand :-
  ntrans(A,ngnd,Q,X,Y),
  ntrans(B,Q,O,_,_),
  ptrans(A,nvdd,O,_,_),
  ptrans(B,nvdd,O,_,_),
  not_connected([Q,O,nvdd,ngnd]),
  not_connected([Q,A]),
  not_connected([Q,B]),
  remove_n(A,ngnd,Q),
  remove_n(B,Q,O),
  remove_p(A,nvdd,O),
  remove_p(B,nvdd,O),
  asserta(nand([A,B],O,X,Y)),
  fail.
nand :-
  ntrans(A,ngnd,Q,X,Y),
  ptrans(A,nvdd,O,_,_),
  ptrans(B,nvdd,O,_,_),
  ntrans(B,Q,R,_,_),
  ntrans(C,R,O,_,_),
  ptrans(C,nvdd,O,_,_),
  not_connected([Q,R,O,nvdd,ngnd]),
  not_connected([Q,R,A]),
  not_connected([Q,R,B]),
  not_connected([Q,R,C]),
  remove_n(A,ngnd,Q),
  remove_n(B,Q,R),
  remove_n(C,R,O),
  remove_p(A,nvdd,O),
  remove_p(B,nvdd,O),
  remove_p(C,nvdd,O),
  asserta(nand([A,B,C],O,X,Y)),
  fail.
nand.
```

For NOR gates, the process is relatively the same as for NAND gates. Definition 11 provides a description of the NOR gate.

Definition 11 The predicate 'nor' is defined in terms of the predicates 'ptrans','ntrans', and 'not_connected' by the implication

$$\begin{aligned}\forall A, B, O, Q, X, Y\ [(&ptrans(A, nvdd, Q, X, Y) \wedge ptrans(B, Q, O, \_, \_)\\ &\wedge ntrans(A, ngnd, O, \_, \_) \wedge ntrans(B, ngnd, O, \_, \_)\\ &\wedge not\_connected([Q, O, nvdd, ngnd]) \wedge not\_connected([Q, A])\\ &\wedge not\_connected([Q, B]))\\ &\Rightarrow\ nor([A, B], O, X, Y)].\end{aligned}$$

The corresponding Prolog extraction rule for a two-input NOR gate is

```
nor :-
  ptrans(A,nvdd,Q,X,Y),
  ptrans(B,Q,O,_,_),
  ntrans(A,ngnd,O,_,_),
  ntrans(B,ngnd,O,_,_),
  not_connected([Q,O,nvdd,ngnd]),
  not_connected([Q,A]),
  not_connected([Q,B]),
  remove_p(A,nvdd,Q),
  remove_p(B,Q,O),
  remove_n(A,ngnd,O),
  remove_n(B,ngnd,O),
  asserta(nor([A,B],O,X,Y,1)),
  fail.
nor.
```

As with the case for the NAND gate, a three-input NOR gate may also be formed.

4.5 Level-N Prolog Rules

The purpose of this section is to describe the higher level rules used to extract leftover transistors and components above level-2. Extraction of leftover transistors is performed using *ad hoc* rules. The *ad hoc* rules are used to identify transistor networks describing certain logical functions that do not conform to discrete CMOS gates. Other extraction rules at this level are formed to identify components such as D flip-flops, registers, half adders, memories, ALUs, etc. The extraction process is performed in as hierarchical fashion as possible. The reason for forming levels of hierarchy for extraction is explained in the following section, which considers the complexity of rules and the efficiency of the extraction-process.

Consider a D flip-flop constructed from an inverter, high-impedance inverter, and transmission gate.

Definition 12 The predicate 'dff' is defined in terms of the predicates 'tgate', 'inv', 'invZ', and 'not_connected' by the implication

$$\begin{aligned}\forall C1, C1bar, Q, I, O, X, Y\ [(&tgate(C1bar, C1, I, Q, X, Y)\\ &\wedge inv(Q, O, \_, \_) \wedge invZ(C1, C1bar, O, Q, \_, \_)\\ &\wedge not\_connected([Q, O]) \wedge not\_connected([Q, I])\\ &\wedge not\_connected([Q, C1, C1bar]))\\ &\Rightarrow\ dff(C1, C1bar, I, O, X, Y)].\end{aligned}$$

The Prolog extraction rule for extracting a D flip-flop is:

```
dff :-
  tgate_rule(C1bar,C1,I,Q,X,Y),
  invZ(C1,C1bar,O,Q,_,_),
  inv(Q,O,_,_),
  not_connected([Q,O]),
  not_connected([Q,I]),
  not_connected([Q,C1,C1bar]),
  remove_tgate(C1bar,C1,I,Q,X,Y),
  retract(invZ(C1,C1bar,O,Q,_,_)),
  retract(inv(Q,O,_,_)),
  asserta(dff(C1bar,C1,D,G,X,Y)),
  fail.
dff.
```

Figure 1 shows a schematic and symbol for a D flip-flop. The schematic is constructed using the components derived from the Level-1 rules.

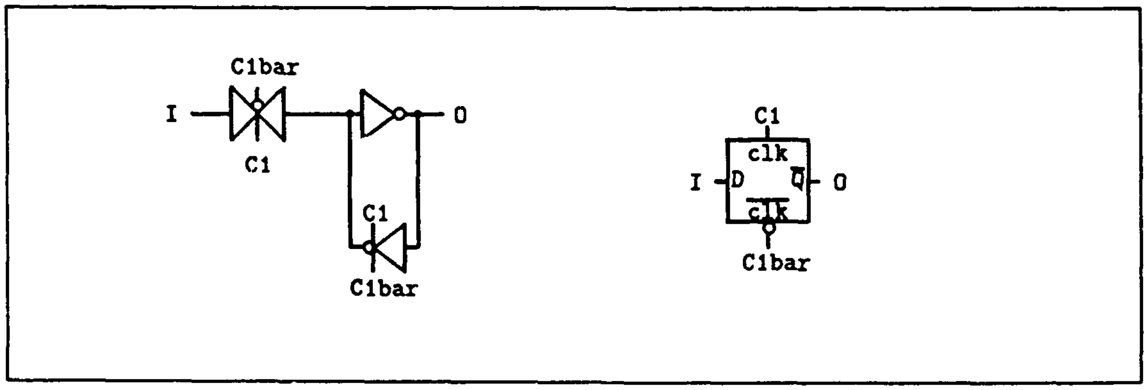

Figure 1. Schematic and Symbol of a D Flip-Flop.

We can see from Definition 12 that knowledge of the lower level transistors is not necessary. We can draw upon already existing knowledge concerning previously extracted components. Definition

12 also demonstrates the ease of identifying the existence of a D flip-flop using already existing extracted components. Using this knowledge in establishing the extraction methodology later will prove beneficial toward program execution efficiency and writing less complicated code.

4.6 *Extraction Heuristics*

The extraction process can occupy a large amount of CPU time. In order to help reduce this time, some heuristics are offered. The first heuristic identifies low-level signature components of higher level components. The second heuristic eliminates duplicate transistors where possible. The last heuristic seeks to reduce the complexity of extraction-rules. These three heuristics are explained below.

4.6.1 Signature Components The goals within the body of a Prolog rule are executed sequentially. Thus, the order in which such goals appear can affect execution efficiency[1]. This procedural aspect of Prolog will be investigated below to help understand how it can be used to speed up extraction. Assume that we are interested in increasing the execution efficiency of a particular Prolog rule, called $comp_1$ shown below.

$$\begin{array}{l} comp_1 \text{ :-} \\ \quad subcompA(A_1, A_2, \ldots, A_o), \\ \quad subcompB(B_1, B_2, \ldots, B_p), \\ \quad subcompC(C_1, C_2, \ldots, C_q), \\ \quad retract(subcompA(A_1, A_2, \ldots, A_o)), \\ \quad retract(subcompB(B_1, B_2, \ldots, B_p)), \\ \quad retract(subcompC(C_1, C_2, \ldots, C_q)), \\ \quad asserta(comp_1(Co_1, Co_2, \ldots, Co_r)), \\ \quad fail. \\ comp_1. \end{array}$$

Assume that there are j *subcompA* components, k *subcompB* components, and l *subcompC* components. Should execution of the $comp_1$ rule lead to m $comp_1$ components exhausting all j, k, and l subcomponents, respectively, then $j = k = l = m$. Assume, now, the existence of another component, $comp_2$, with the rule

$$\begin{array}{l} comp_2 \text{ :-} \\ \quad subcompB(B_1, B_2, \ldots, B_p), \\ \quad subcompC(C_1, C_2, \ldots, C_q), \\ \quad retract(subcompB(B_1, B_2, \ldots, B_p)), \\ \quad retract(subcompC(C_1, C_2, \ldots, C_q)), \\ \quad asserta(comp_2(Co_1, Co_2, \ldots, Co_s)), \\ \quad fail. \\ comp_2. \end{array}$$

Assume, too, that n $comp_2$ components exist and that after the application of both rules, $comp_1$ and $comp_2$, all j, k, and l subcomponents will be exhausted. From the above extraction process, then, $j = m$ and $k = l = m + n$. Since *subcompA* occurs only in $comp_1$, *subcompA* is called the signature of $comp_1$.

[1]For those more familiar with Prolog, the terms *shallow* and *deep* backtracking may come to mind. For those wishing to know more about *shallow* and *deep* backtracking, a discussion is found in [7]

Furthermore, extraction using the Prolog rule *comp*$_1$ before the Prolog rule *comp*$_2$ is preferred. If we were to use the Prolog rule *comp*$_2$ first, then all k *subcompB* components would be searched for inclusion in *comp*$_2$. Whereas, using the Prolog rule *comp*$_1$ first would reduce the search space to $k - m$ *subcompB* components for *comp*$_2$. The search rationale is further predicated on the assumption that some of the parameters for a subcomponent will aid in the selection of subsequent subcomponents in a Prolog rule. To help understand how parameters aid in the selection of subsequent subcomponents, consider the following explanation.

Assume that some parameter of *subcompA*, called A_h where $1 \leq h \leq o$, is connected to some parameter of *subcompB*, called B_i where $1 \leq i \leq p$, in *comp*$_1$. In the execution of the Prolog rule *comp*$_1$, Prolog will attempt to find a component in the component database called *subcompA* before looking for *subcompB*. Once a component is found satisfying *subcompA*, the parameters of *subcompA* will be instantiated (or unified) to the values corresponding to the component in the component database. Since $B_i = A_h$ in *subcompB*, B_i is instantiated to the value of A_h and will therefore constrain Prolog to finding a component that satisfies *subcompB* and B_i. The additional constraint of B_i reduces the possible components to be considered in satisfying *comp*$_1$.

Consider the following example using the Prolog rule described earlier for a D flip-flop,

```
dff :-
  clk_inv(P2,P1,G,X,Xloc,Yloc,1),
  tgate(P1,P2,D,X,_,_),
  inv(X,G,_,_,1),
  remove_tgate(P1,P2,D,X),
  retract(clk_inv(P2,P1,G,X,Xloc,Yloc,1)),
  retract(inv(X,G,_,_,1)),
  asserta(dff(P1,P2,D,G,Xloc,Yloc,1)),
  fail.
dff.
```

We may compare the `dff` rule to a new rule for an `Exclusive-OR`, called `xor`, below.

```
xor :-
  tgate(B,Bnot,A,XOR,_,_),
  inv(B,Bnot,_,_,1),
  inv(A,Anot,_,_,1),
  tgate(Bnot,B,Anot,XOR,Xloc,Yloc),
  retract(inv(B,Bnot,_,_,1)),
  retract(inv(A,Anot,_,_,1)),
  remove_tgate(Bnot,B,Anot,XOR),
  remove_tgate(B,Bnot,A,XOR),
  asserta(xor(A,Anot,B,Bnot,XOR,Xloc,Yloc,3)),
  fail.
xor.
```

Both `dff` and `xor` share transmission gates and inverters; however, `clk_inv` only occurs in `dff`. In this case, `clk_inv` would be considered a signature component for `dff`. Also notice in `xor` that finding `tgate(B,Bnot,A,XOR,_,_)` would easily lead to location of one inverter, aid in the quick selection of a second, and to location of the second transmission gate.

4.6.2 Eliminating Duplicates The second heuristic seeks to eliminate duplicate transistors. Since only digital logic is of interest, additional transistors (added to increase the drive capacity of a circuit) needlessly increase the search space. When only looking for the logic functionality of a circuit, no additional information is gained from such transistors. The following is a Prolog rule adopted to eliminate duplicate transistors while reading in the transistor netlist from a mask layout description.

```
remove_dup_trans :-
  read(X),
  remove_dup_trans(X),!,
  remove_dup_trans.
remove_dup_trans.
remove_dup_trans(end_of_file) :- !.
remove_dup_trans(p(A,B,C,_,_,_,_)) :-
  ptrans(A,B,C,_,_),!.
remove_dup_trans(n(A,B,C,_,_,_,_)) :-
  ntrans(A,B,C,_,_),!.
remove_dup_trans(p(A,B,C,W,L,X,Y)) :-
  asserta(p(A,B,C,W,L,X,Y)),!.
remove_dup_trans(n(A,B,C,W,L,X,Y)) :-
  asserta(n(A,B,C,W,L,X,Y)),!.
```

4.6.3 Reducing Prolog Rule Complexity The third heuristic addresses rule complexity. Rule complexity is directly related to the number of components that must be matched. In general, simpler rules increase execution efficiency.

An example of how rule complexity influences efficiency may be found in the identification of registers from a component netlist. Assume the following rule, *register1*, for registers.

```
register1 :-
  clk_inv(R,P,C1,C1bar,X,Y),
  inv(P,R,_,_),
  tgate(In,P,C1bar,C1,_,_),
  tgate(R,Q,C2bar,C2,_,_),
  clk_inv(S,Q,C2,C2bar,_,_),
  inv(Q,S,_,_),
  tgate(S,Out,Abar,A,_,_),
  retract(clk_inv(R,P,C1,C1bar,X,Y)),
  retract(inv(P,R,_,_)),
  retract(tgate(In,P,C1bar,C1,_,_)),
  retract(tgate(R,Q,C2bar,C2,_,_)),
  retract(clk_inv(S,Q,C2,C2bar,_,_)),
  retract(inv(Q,S,_,_)),
  retract(tgate(S,Out,Abar,A,_,_)),
  asserta(register(In,Out,C1,C1bar,C2,C2bar,A,Abar,X,Y)),
  fail.
register1.
```

Figure 2 is a diagram of the component extracted by the *register1* rule.

Assume a component netlist consisting of *clk_inv*, *inv*, and *tgate* that when extracted form *j* registers with no residual components. Assume also that a register is constructed from two D flip-flops, described below, and a transmission gate as in Figure 3 and by the Prolog rule for *register2* that follows.

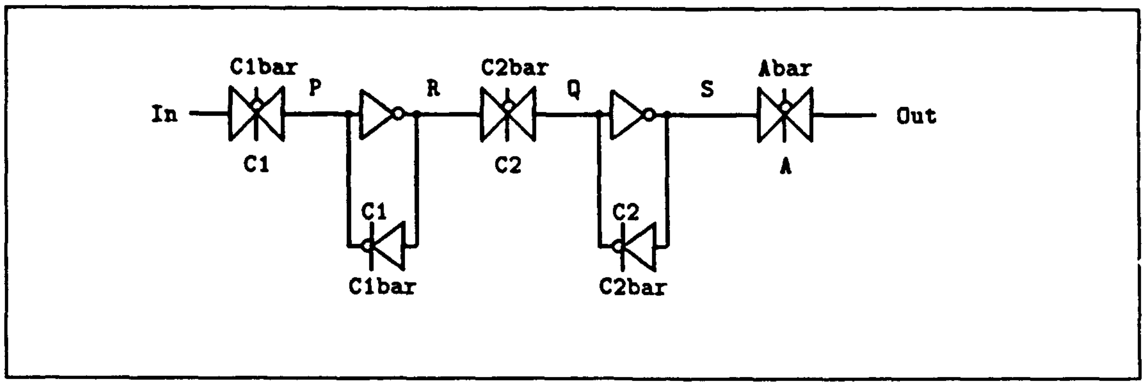

Figure 2. Schematic for *register1* Extraction Rule.

```
register2 :-
  dff(In,R,C1,C1bar,X,Y),
  dff(R,S,C2,C2bar,_,_),
  tgate(S,Out,Abar,A,_,_),
  retract(dff(In,R,C1,C1bar,X,Y)),
  retract(dff(R,S,C2,C2bar,_,_)),
  retract(tgate(S,Out,Abar,A,_,_)),
  asserta(register(In,Out,C1,C1bar,C2,C2bar,A,Abar,X,Y)),
  fail.
register2.

dff :-
  clk_inv(R,P,C1,C1bar,X,Y),
  inv(P,R,_,_),
  tgate(In,P,C1bar,C1,_,_),
  retract(clk_inv(R,P,C1,C1bar,X,Y)),
  retract(inv(P,R,_,_)),
  retract(tgate(In,P,C1bar,C1,_,_)),
  asserta(dff(In,R,C1,C1bar,X,Y)),
  fail.
dff.
```

Using the rules *register2* and *dff* there are *k dff*, where $k = 2 * j$, and *j tgate*. If the rule *register1* is considered, there are *k clk_inv*, *k inv*, and *l tgate* where $l = j + k$.

For the purpose of the illustration consider *register1*, *dff*, and *register2* in the following manner. The parts of each rule that query the fact database will also be numbered to aid in the discussion.

```
        register1 :-
R11       clk_inv(R,P,C1,C1bar,X,Y),
R12       inv(P,R,_,_),
R13       tgate(In,P,C1bar,C1,_,_),
R14       tgate(R,Q,C2bar,C2,_,_),
R15       clk_inv(S,Q,C2,C2bar,_,_),
R16       inv(Q,S,_,_),
```

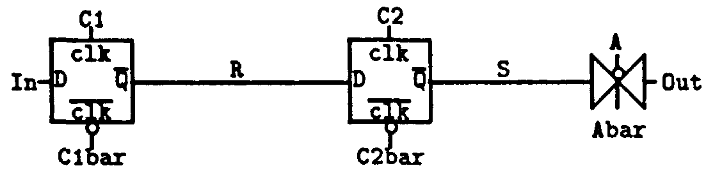

Figure 3. A Second Schematic for *register2* Extraction Rule.

```
R17          tgate(S,Out,Abar,A,_,_),

           dff :-
D1           clk_inv(R,P,C1,C1bar,X,Y),
D2           inv(P,R,_,_),
D3           tgate(In,P,C1bar,C1,_,_),

           register2 :-
R21          dff(In,R,C1,C1bar,X,Y),
R22          dff(R,S,C2,C2bar,_,_),
R23          tgate(S,Out,Abar,A,_,_),
```

Statements R11, D1, and R21 may be considered as enumeration statements (or ENUMERATE) since they simply pick off from the database of facts the next available fact until all facts that satisfy predicate/arity have been exhausted. Statements R12...R17, D2, D3, R22, and R23, may be considered as database queries (or QUERY) since some or all of their parameters have been unified based upon the previous statements. If we also assume the worst-case ordering of components such that the first *k* *clk_inv* actually form the second *dff*, the rule *register1* will "fail" *k* times before it will actually begin identifying registers. Furthermore, the *k* times that *register1* failed it identified *k* *dff*. Using *register1* to identify registers, there will be at most *k* failed ENUMERATEs and $4 * k$ failed QUERYs. The failed QUERYs are incurred since R12, R13, and R14 succeed, but R15 will fail causing the entire sequence to backtrack and try a new *clk_inv*.

Consider the rules *dff* and *register2* on the same component netlist. The rule *dff* will succeed until all *clk_inv* have been exhausted. If we assume that the *dff* components were ordered in the worst case then *register2* will have only *k* failed ENUMERATEs and no more. The $4 * k$ failed QUERYs from *register1* were avoided by reducing its complexity.

Generally, the above three heuristics have been found to increase the speed of execution. Identifying signature components may be dependent on the composition of a given component netlist and should therefore be considered. Eliminating duplicate components not only reduces the search space but allows for parallelization of the extraction process. Finally, reducing rule complexity increases efficiency by reducing search failures.

4.7 *Generating an Output File*

Two methods exist for generating an output file from a component netlist in Prolog. The first one was hinted at earlier and used the `listing/1` Prolog function. The second method uses Prolog rules to generate a file or group of files containing the components in the Prolog database.

Using the `listing/1` Prolog function will print out the fact base. Use of the `listing/1` Prolog function can be demonstrated through an example. First a file, `trans.pro`, containing some Prolog extraction rules will be read into Quintus Prolog using the `compile/1` Prolog function.

```
% prolog

Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- compile(['trans.pro']).
[compiling /usr/users/dukes/class/trans.pro...]
[trans.pro compiled 1.284 sec 1,424 bytes]

yes
| ?-
```

Another file, `intrans.pro`, containing a transistor netlist will be read into the Prolog fact base.

```
| ?- ['intrans.pro'].
[consulting /usr/users/dukes/class/intrans.pro...]
[intrans.pro consulted 0.183 sec 944 bytes]

yes
| ?-
```

To inspect the Prolog fact base, the `listing/0` Prolog function will be used.

```
| ?- listing.

library_directory('/usr/local/q2.4/library').
library_directory('/usr/local/q2.4/tools').
library_directory('/usr/local/q2.4/IPC').

p(nGATED,nALPHA,nBETA,3,6,783,132).
p(nRIGHT_SELECT,n30_21_FILL,n30_21_FILLOUT,3,6,20985,23).
p(nINPUT,nvdd,nINTER1,3,6,1254,387).
p(nSELECT,nINTER1,nOUTPUT,3,6,1254,387).

n(nSELECTBAR,nINTER2,nOUTPUT,3,6,39887,-3091).
n(nINPUT,ngnd,nINTER2,3,6,1260,387).
```

```
yes
| ?-
```

Aside from the **library_directory** statements, we can see that a list of p-type MOS transistors and a list of n-type MOS transistors are present. At this point, the Prolog extraction rule for `invZ/0` is entered.

```
| ?- invZ.

yes
| ?-
```

By entering the Prolog function `listing/0`, we can see the Prolog fact base as it exists after executing `invZ`.

```
| ?- listing.

library_directory('/usr/local/q2.4/library').
library_directory('/usr/local/q2.4/tools').
library_directory('/usr/local/q2.4/IPC').

p(nGATED,nALPHA,nBETA,3,6,783,132).
p(nRIGHT_SELECT,n30_21_FILL,n30_21_FILLOUT,3,6,20985,23).

invZ(nSELECT,nSELECTBAR,nINPUT,nOUTPUT,1254,387).

yes
| ?-
```

The procedure of using `listing/0` may be automated by setting up a command file. A possible command, `pro.com`, is shown below.

```
compile(['trans.pro']).
['intrans.pro'].
invZ.
listing.
halt.
```

The command file is executed using `csh` in the following manner.

```
% prolog<pro.com>&pro.log
```

The component netlist has been captured within the file pro.log. The pro.log file contains the following:

```
% cat pro.log

Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

[compiling /usr/users/dukes/class/trans.pro...]
[trans.pro compiled 1.266 sec 1,424 bytes]

yes
[consulting /usr/users/dukes/class/intrans.pro...]
[intrans.pro consulted 0.184 sec 944 bytes]

yes

yes

library_directory('/usr/local/q2.4/library').
library_directory('/usr/local/q2.4/tools').
library_directory('/usr/local/q2.4/IPC').

p(nGATED,nALPHA,nBETA,3,6,783,132).
p(nRIGHT_SELECT,n30_21_FILL,n30_21_FILLOUT,3,6,20985,23).

invZ(nSELECT,nSELECTBAR,nINPUT,nOUTPUT,1254,387).

yes
```

The file, pro.log, may be edited to remove the extraneous lines.

The second method of generating a file of extracted and unextracted components is performed using tell/1, write/1, nl/0, and told/0. tell/1 opens a file passed as an argument. write/1 outputs its argument to a file opened by tell/1. nl/0 generated a carriage return and line feed in the file opened by tell/1. Finally, told/0 closes the file opened by tell/1.

In a file called list.pro, Prolog rules using write/1 and nl/0 will be placed for writing out the transistors left over in the facts base. The contents of the file are

```
list_trans :-
  retract(p(A,B,C,W,L,X,Y)),
  write(p(A,B,C,W,L,X,Y)),write('.'),nl,
  fail.
list_trans :-
  retract(n(A,B,C,W,L,X,Y)),
  write(n(A,B,C,W,L,X,Y)),write('.'),nl,
  fail.
```

```
list_trans.
```

The Prolog rule, `list_trans/0`, retracts a transistor from the facts base. Afterwards, the transistor is written out followed by a carriage return and line feed. If there are no further p-type or n-type transistors left in the facts database, the last statement, `'list_trans.'`, simply allows the Prolog rule to be satisfied.

To see how this works, a session using Prolog has been provided below. All that is being performed in this case is to load in the transistors from `intrans.pro`, open a new file called `outtrans.pro`, write the transistors out to the new file, close the file `outtrans.pro`, and halt the Prolog session.

```
% prolog
Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- compile(['list.pro']).
[compiling /usr/users/dukes/class/list.pro...]
[list.pro compiled 0.367 sec 584 bytes]

yes
| ?- ['intrans.pro'].
[consulting /usr/users/dukes/class/intrans.pro...]
[intrans.pro consulted 0.200 sec 992 bytes]

yes
| ?- tell('outtrans.pro').

yes
| ?- list_trans.

yes
| ?- told.

yes
| ?- halt.
%
```

Examining both `intrans.pro` and `outtrans.pro`, we can see that there is no difference except for the first line in `intrans.pro` which is a Prolog command.

```
% cat intrans.pro
:- dynamic p/7,n/7.
p(nGATED,nALPHA,nBETA,3,6,783,132).
p(nRIGHT_SELECT,n30_21_FILL,n30_21_FILLOUT,3,6,20985,23).
p(nINPUT,nvdd,nINTER1,3,6,1254,387).
```

```
p(nSELECT,nINTER1,nOUTPUT,3,6,1254,387).
n(nSELECTBAR,nINTER2,nOUTPUT,3,6,39887,-3091).
n(nINPUT,ngnd,nINTER2,3,6,1260,387).
% cat outtrans.pro
p(nGATED,nALPHA,nBETA,3,6,783,132).
p(nRIGHT_SELECT,n30_21_FILL,n30_21_FILLOUT,3,6,20985,23).
p(nINPUT,nvdd,nINTER1,3,6,1254,387).
p(nSELECT,nINTER1,nOUTPUT,3,6,1254,387).
n(nSELECTBAR,nINTER2,nOUTPUT,3,6,39887,-3091).
n(nINPUT,ngnd,nINTER2,3,6,1260,387).
```

V. Finding Design Errors

5.1 Introduction

The extraction methodology previously described has only been concerned with extracting normal circuits. However, we cannot assume that the circuit to be extracted is free from design errors. Since errors may exist in a design, we must be prepared to find them. There are two types of interconnection errors that can be identified rather easily in Prolog. The division between the two types occurs at the component boundary. The first error occurs because the external interconnections of a component are configured in an inconsistent condition. The second error occurs because an internal node of a component is connected to another component. In this section, the problem of identifying these errors will be discussed.

5.2 Identifying External Design Errors

The CMOS designs described earlier were based on a design style of a particular designer or group of designers. Just as CMOS designs are based on a design style, so too are design errors. A few of the types of design errors possible are shown in Figure 4. It is important to point out that the following circuits are considered to be errors because they are not normally used in the design of a VLSI circuit.

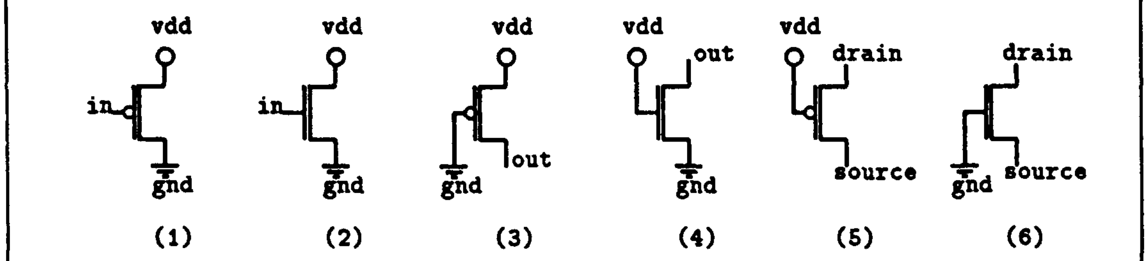

Figure 4. Some Common Design Flaws.

Subfigures 1 and 2 of Figure 4 typify a dangerous circuit. This type of design error may result from one of several actions on the part of the layout system used. If plowing or some other form of circuit rearrangement is being performed, it is possible to connect the terminals of the transistor in the fashion shown. During layout in *magic*, routing over subcells with metal layers that accidentally contact the same layer of lower subcells may also cause this problem. In either case, the result is a circuit that, when turned on, will cause a short.

The circuits in subfigures 3 and 4 of Figure 4 are a little less destructive than the circuits discussed earlier. However, they may indicate design errors. These circuits may be easily replaced by a metal line connected to Vdd for subfigure 3 or GND for subfigure 4. Even though these circuits may be caused by the problems indicated for subfigures 1 and 2, they may also be the result of tying inputs to arrays of standard cells high or low. Subfigures 5 and 6 of Figure 4 demonstrate another possible design error. As with the circuits in subfigures 3 and 4, their creation may be accidental or incidental. The following "error" definition is offered.

Definition E1 $\forall In, Out, Drain, Source,$

1. ptrans(In,vdd,gnd) is an error;
2. ntrans(In,vdd,gnd) is an error;
3. ptrans(gnd,vdd,Out) is an error;
4. ntrans(vdd,Out,gnd) is an error;
5. ptrans(vdd,Drain,Source) is an error;
6. ntrans(gnd,Drain,Source) is an error.

Recognition of design flaws is not limited to single transistors. Erroneous designs consisting of groups of transistors may also be considered. Figure 5 provides examples of designs that may be considered design errors. For these structures, another "error" definition may be considered.

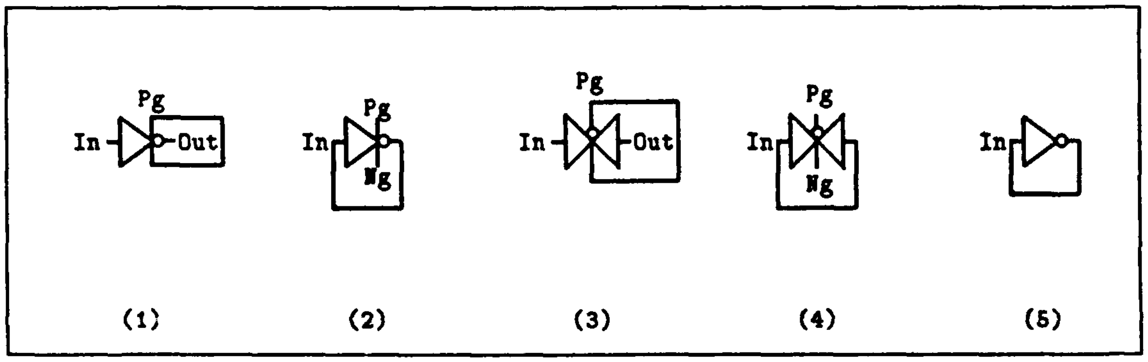

Figure 5. Some Common Design Errors.

Definition E2 $\forall Pg, Ng, In, Out,$

1. invZ(Pg,Pg,In,Out) is an error;
2. invZ(Pg,Ng,In,In) is an error;
3. tgate(Pg,Pg,In,Out) is an error;
4. tgate(Pg,Ng,In,In) is an error;
5. inv(In,In) is an error.

The list of errors included in Definitions E1 and E2 is not complete. For some designs, some of the enumerated errors may not be errors at all. Therefore, definitions for design errors are declared within the domain of the design style under consideration.

5.3 Prolog Implementation for Identifying External Design Errors

This section describes two methods for using Prolog to identify design errors. The first method is an interactive one where the user provides statements to be satisfied by Prolog from the component database. The second method allows the user to specify a list of Prolog rules that may be stored in a file and executed at a later time.

Definitions E1 and E2 designate certain component configurations to be erroneous. Using Prolog interactively, these errors may be identified easily. The following is a demonstration of how the component database is examined for the occurrence of the first type of error in Definition E1.

```
| ?- ptrans(In,nvdd,ngnd,X,Y).
In = n20_024_14A_BAR,
X = 5722,
Y = 141 ;
In = n20_024_13A_BAR,
X = 5722,
Y = 506 ;
In = n20_024_12A_BAR,
X = 5722,
Y = 798 ;
In = n20_024_11A_BAR,
X = 5722,
Y = 1017
```

Notice the addition of the two additional fields, X and Y. These fields contain location information that may be used to find the errant components. These fields were added to all components. Through the use of Definition 3, we were able to ask Prolog to identify those transistors that satisfied one of the design error types in Definition E1. We may also perform the same query for higher level components as shown below.

```
| ?- tgate(Pg,Ng,In,In,X,Y,1).
no
| ?- tgate(Pg,Pg,In,Out,X,Y,1).
Pg = n12_0typeIIId_4XOR,
In = n12_0typeIIId_4COUT1,
Out = n12_0typeIIIb_15COUT1,
X = -306,
Y = 97 ;
Pg = n12_0typeIIIb_15XOR,
In = n12_0typeIIIb_15COUT1,
Out = n12_0typeIIIa_14COUT1,
X = -306,
Y = 170 ;
Pg = n12_0typeIIIc_9XOR,
In = n12_0typeIIIc_9COUT1,
Out = n12_0typeIIIb_14COUT1,
X = -290,
Y = 316
```

An additional field was incorporated in the higher level components to distinguish the original design style used to create a component. In this particular case, the 1 denotes CMOS design style. Had we been interested in NMOS design style, a 2 would be used in this field.

In the previous example, the component database was queried for the existence of any transmission gates that met condition 4 of Definition E2. In this case, no transmission gates were found.

However, when Prolog was queried for the existence of transmission gates that violated condition 3 of Definition E2, several instances were found and reported.

Design errors may also be found through the establishment of Prolog rules prior to performing duplicate transistor reduction and extraction in the case of Definition E1. The following is an example of a rule used to find a design error identified in Definition E1.

```
/*  Error type 1   */
find_error :-
  ptrans(G,nvdd,ngnd,X,Y),
  write('Bad trans, '),write(ptrans(G,nvdd,ngnd,X,Y)),
  write(': removed'),nl,
  remove_p(G,nvdd,ngnd),
  fail.
/*  Error type 2   */
find_error :-
  ntrans(G,nvdd,ngnd,X,Y),
  write('Bad trans, '),write(ntrans(G,nvdd,ngnd,X,Y)),
  write(': removed'),nl,
  remove_n(G,nvdd,ngnd),
  fail.
/*  Error type 3   */
find_error :-
  ptrans(ngnd,nvdd,S,X,Y),
  write('Straight wire, '),write(ptrans(ngnd,nvdd,S,X,Y)),
  write(': removed'),nl,
  remove_p(ngnd,nvdd,S),
  fail.
/*  Error type 4   */
find_error :-
  ntrans(nvdd,ngnd,S,X,Y),
  write('Straight wire, '),write(ntrans(nvdd,ngnd,S,X,Y)),
  write(': removed'),nl,
  remove_n(nvdd,ngnd,S),
  fail.
/*  Error type 5   */
find_error :-
  ptrans(nvdd,A,B,X,Y),
  write('Open connection, '),write(ptrans(nvdd,A,B,X,Y)),
  write(': removed'),nl,
  remove_p(nvdd,A,B),
  fail.
/*  Error type 6   */
find_error :-
  ntrans(ngnd,A,B,X,Y),
  write('Open connection, '),write(ntrans(ngnd,A,B,X,Y)),
  write(': removed'),nl,
  remove_n(ngnd,A,B),
  fail.
find_error.
```

Notice that `find_error.` is listed last. This is to provide a successful outcome when all of the previous clauses fail.

The following Prolog rules are used to identify design errors that conform to Definition E2.

```
find_more_errors :-
  clk_inv(P,P,In,Out,X,Y,1),
  write('Screwy clk_inv, '),write(clk_inv(P,P,In,Out,X,Y,1)),
  write(': removed'),nl,
  retract(clk_inv(P,P,In,Out,X,Y,1)),
  fail.
find_more_errors :-
  clk_inv(Pg,Ng,Bad,Bad,X,Y,1),
  write('Oscillating clk_inv, '),write(clk_inv(Pg,Ng,Bad,Bad,X,Y,1)),
  write(': removed'),nl,
  retract(clk_inv(Pg,Ng,Bad,Bad,X,Y,1)),
  fail.
find_more_errors :-
  tgate(P,P,In,Out,X,Y,1),
  write('Screwy tgate, '),write(tgate(P,P,In,Out,X,Y,1)),
  write(': removed'),nl,
  retract(tgate(P,P,In,Out,X,Y,1)),
  fail.
find_more_errors :-
  tgate(Pg,Ng,Bad,Bad,X,Y,1),
  write('Worthless tgate, '),write(tgate(Pg,Ng,Bad,Bad,X,Y,1)),
  write(': removed'),nl,
  retract(tgate(Pg,Ng,Bad,Bad,X,Y,1)),
  fail.
find_more_errors :-
  inv(Bad,Bad,X,Y,1),
  write('Oscillating inv, '),write(inv(Bad,Bad,X,Y,1)),
  write(': removed'),nl,
  retract(inv(Bad,Bad,X,Y,1)),
  fail.
find_more_errors.
```

5.4 Identifying Internal Design Errors

Identifying internal design errors is performed as a type of insurance policy. Essentially, we wish to check that an internal node of a component is free from any connections outside the component. Figure 6 shows the general case for an internal design error. If $Comp_1$ was extracted without consideration for the external connection, the result would be lost information about the connectivity of $Comp_1$ to $Comp_2$. We must determine whether the lost information is a proper connection and include it as a parameter for the extracted $Comp_1$, or whether the lost information is really a design error that should be identified.

Figure 7 is a more specific example of an internal design error involving two D flip-flops. We see that the internal node of the D flip-flop is connected to an input of another D flip-flop. Normally, a proper design of a D flip-flop would not include a tap off the internal node. This type of internal connection places additional capacitive loading which can cause a longer rise or fall time and perturb the timing performance of the circuit.

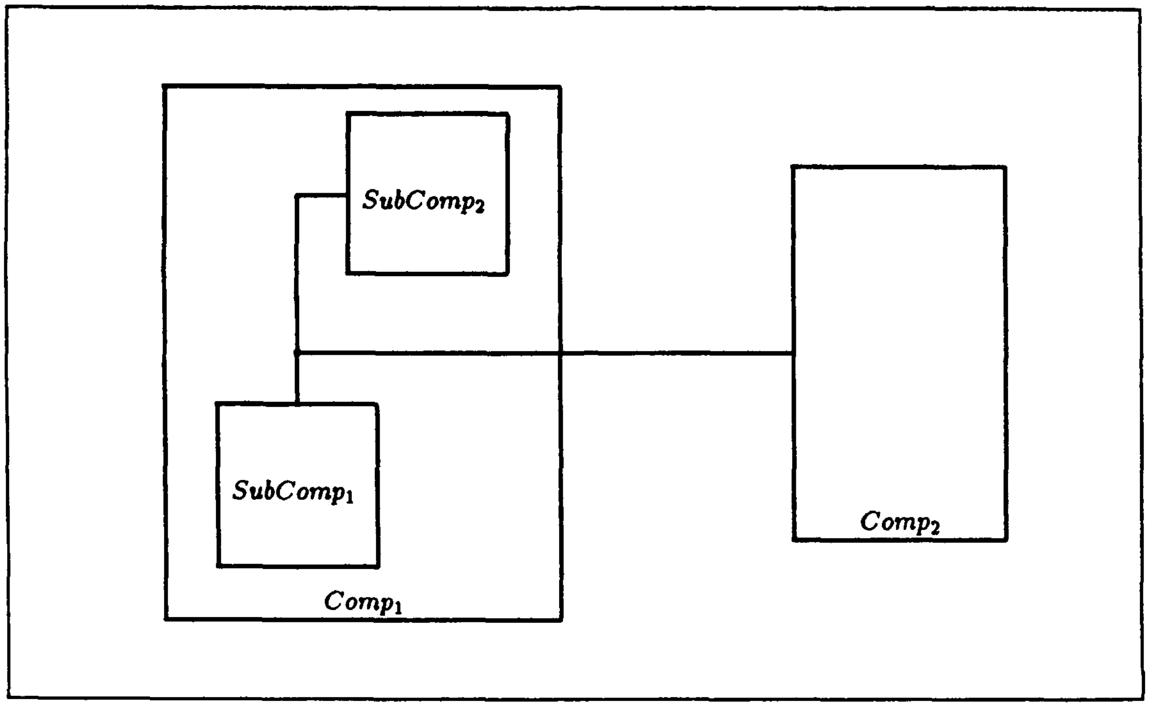

Figure 6. General Type of Internal Design Error.

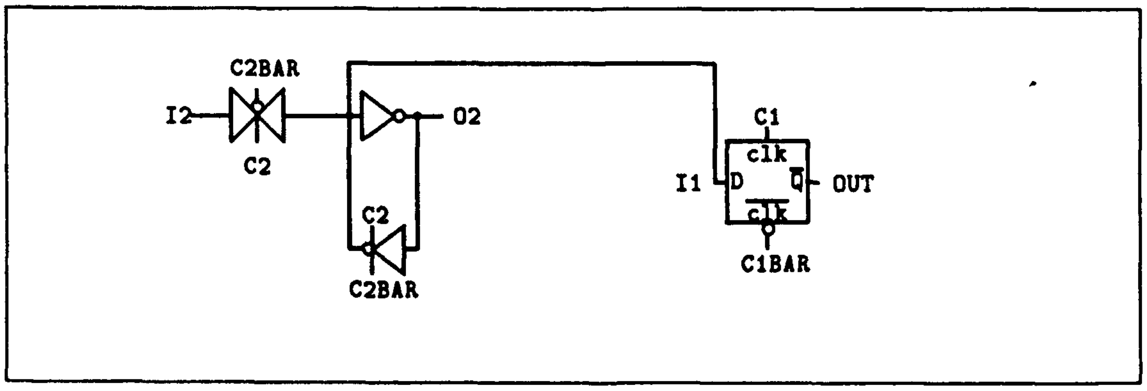

Figure 7. Internal Design Error of a D Flip-Flop.

If we use the following D flip-flop extraction rule to finish extracting D flip-flops, Figure 8 is the result.

```
dff :-
  tgate_rule(C1bar,C1,I,Q,X,Y),
  invZ(C1,C1bar,O,Q,_,_),
  inv(Q,O,_,_),
  not_connected([Q,O]),
  not_connected([Q,I]),
  not_connected([Q,C1,C1bar]),
  remove_tgate(C1bar,C1,I,Q,X,Y),
  retract(invZ(C1,C1bar,O,Q,_,_)),
  retract(inv(Q,O,_,_)),
  asserta(dff(C1bar,C1,D,G,X,Y)),
  fail.
dff.
```

From Figure 8 we can see that connection information has been lost. Furthermore, information about a possible design error has been lost.

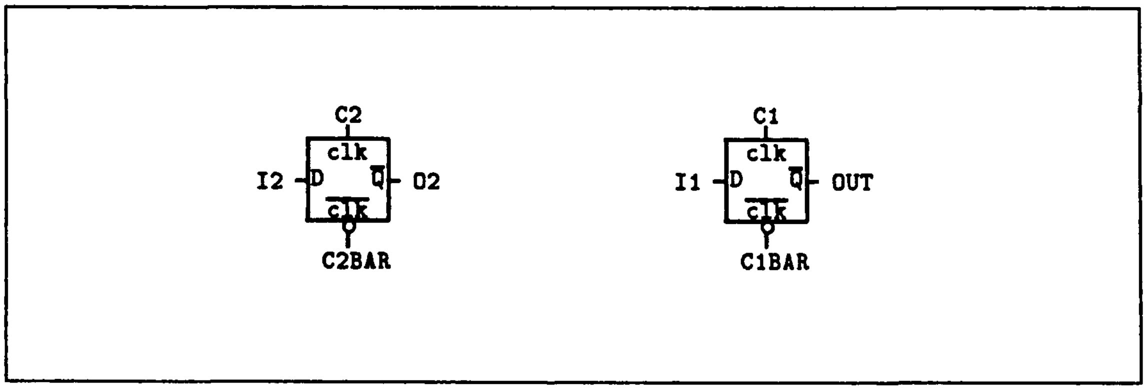

Figure 8. Result of Extracting an Internal Design Error of a D Flip-Flop.

5.5 *Prolog Implementation for Identifying Internal Design Errors*

This section provides the Prolog rules for identifying internal design errors. Also included is a discussion of how the error identification rules are called within the Prolog extraction rules. At the end of this section is a list of commonly used error-identification Prolog rules that may be included within any Prolog extraction system.

The first set of Prolog rules pertain to finding internal connections to transistors. The Prolog rules for identifying this type of internal design error are:

```
find_anomaly_list(_,[]) :- !.
find_anomaly_list(Comp,[Node|Rest]) :-
  find_anomaly(Comp,Node),!,
  find_anomaly_list(Comp,Rest).
find_anomaly_list(Comp,[_|Rest]) :-
  find_anomaly_list(Comp,Rest).

find_anomaly(Comp,Node) :-
  (ptrans(Node,_,_,X,Y);
   ptrans(_,Node,_,X,Y);
   ntrans(Node,_,_,X,Y);
   ntrans(_,Node,_,X,Y)),
  write('Failure extracting component '),write(Comp),
  write('.'),nl,write('     Internal node, '),write(Node),
  write(', connected to a transistor at X:'),
  write(X),write(', Y:'),write(Y),write('.'),nl.
```

The Prolog rule, **find_anomaly_list/2**, takes as its arguments the name of the extracted component and a list of the internal nodes. Each node and extracted component is then passed to a second Prolog rule, **find_anomaly/2**, to determine if any transistors are connected to the internal node. If a connection is found, the extracted component, the internal node name, the transistor, and the transistor location are reported. The ; in Prolog is used to form a disjunction of goals. The Prolog rule, **find_anomaly/2**, may be improved upon slightly. Recalling that **ptrans/5** and **ntrans/5** are rules that regard the source and drain interchangeable, the goal clauses **ptrans(Node,_,_,X,Y)** and **ntrans(Node,_,_,X,Y)** each test the fact base twice for a transistor whose gate can be unified with the value passed in through **Node**. Therefore, substituting **p(Node,_,_,_,_,X,Y)** for **ptrans(Node,_,_,X,Y)** and **n(Node,_,_,_,_,X,Y)** for **ntrans(Node,_,_,X,Y)** will save two searches on the Prolog fact base. The following shows **find_anomaly/2** rewritten for efficient execution.

```
find_anomaly(Comp,Node) :-
  (p(Node,_,_,_,_,X,Y);
   ptrans(_,Node,_,X,Y);
   n(Node,_,_,_,_,X,Y);
   ntrans(_,Node,_,X,Y)),
  write('Failure extracting component '),write(Comp),
  write('.'),nl,write('     Internal node, '),write(Node),
  write(', connected to a transistor at X:'),
  write(X),write(', Y:'),write(Y),write('.'),nl.
```

VI. Putting Extraction and Error Identification Together

The purpose of this section is to show how extraction and error identification are used together to generate a high-level component netlist from a transistor netlist and report errors within the layout design. This discussion will present both extraction and error identification as one complete system. The system is made up of Prolog input rules, Prolog utility rules, Prolog extraction rules, Prolog error idenitification rules, Prolog output rules, and a main Prolog driver. The code presented in this section has been used to perform extraction on fabricated designs. Thus, users of GES may use the code directly from this section against their CMOS design.

6.1 Prolog Utility Rules

The Prolog rules described in this section are called Prolog utility rules since they are called upon by other Prolog rules within GES. The rules are as follows.

```
ptrans(G,D,S,X,Y) :-
  p(G,D,S,_,_,X,Y).
ptrans(G,D,S,X,Y) :-
  p(G,S,D,_,_,X,Y).

ntrans(G,D,S,X,Y) :-
  n(G,D,S,_,_,X,Y).
ntrans(G,D,S,X,Y) :-
  n(G,S,D,_,_,X,Y).

remove_p(G,D,S) :-
  retract(p(G,D,S,_,_,_,_)),!.
remove_p(G,D,S) :-
  retract(p(G,S,D,_,_,_,_)),!.

remove_n(G,D,S) :-
  retract(n(G,D,S,_,_,_,_)),!.
remove_n(G,D,S) :-
  retract(n(G,S,D,_,_,_,_)),!.

tgate(P,N,I,O,X,Y) :-
  tgate(P,N,I,O,X,Y,1).
tgate(P,N,I,O,X,Y) :-
  tgate(P,N,O,I,X,Y,1).

remove_tgate(P,N,O,I) :-
  retract(tgate(P,N,O,I,_,_,1)),!.
remove_tgate(P,N,O,I) :-
  retract(tgate(P,N,I,O,_,_,1)).

not_connected([]) :- !.
not_connected([Node|Tail]) :-
  not_member(Node,Tail),
  not_connected(Tail).

not_member(_,[]) :- !.
not_member(Node,[Head|Tail]) :-
  Node \== Head,
  not_member(Node,Tail).
```

A discussion of these Prolog rules may be found in Section IV.

6.2 Prolog Input Rules

Outlined in this section is a list of the Prolog input rules used to read in the transistor netlist and eliminate duplicates. The Prolog rules are shown below.

```
remove_dup_trans :-
  read(X),
  remove_dup_trans(X),!,
  remove_dup_trans.
remove_dup_trans.
remove_dup_trans(end_of_file) :- !,fail.
remove_dup_trans(p(A,B,C,_,_,_,_)) :-
  ptrans(A,B,C,_,_),!.
remove_dup_trans(n(A,B,C,_,_,_,_)) :-
  ntrans(A,B,C,_,_),!.
remove_dup_trans(p(A,B,C,W,L,X,Y)) :-
  asserta(p(A,B,C,W,L,X,Y)),!.
remove_dup_trans(n(A,B,C,W,L,X,Y)) :-
  asserta(n(A,B,C,W,L,X,Y)),!.
```

The Prolog rule, **remove_dup_trans/0** works by reading in a record from the input file, checking to see if it is the end-of-file marker, then checking to see if the transistor already exists in the database. If all of the above tests fail, the transistor is added to the database.

6.3 Prolog Extraction Rules

Within this section, only some of the previously-discussed Prolog extraction rules will be listed. A discussion of the Level-1, Level-2, and Level-N Prolog extraction rules may be found in Section IV.

```
/* Level-1 Prolog Extraction Rules */

inv :-
  ptrans(G,D,nvdd,X,Y),
  ntrans(G,ngnd,D,_,_),
  remove_p(G,D,nvdd),
  remove_n(G,ngnd,D),
  asserta(inv(G,D,X,Y,1)),
  fail.
inv.
```

```
tgate :-
  ptrans(G,A,B,X,Y),
  ntrans(H,A,B,_,_),
  remove_p(G,A,B),
  remove_n(H,A,B),
  asserta(tgate(G,H,A,B,X,Y,1)),
  fail.
tgate.

invZ :-
  ptrans(I,nvdd,Q,X,Y),
  ptrans(P,Q,O,_,_),
  ntrans(N,O,R,_,_),
  ntrans(I,R,ngnd,_,_),
  not_connected([Q,R,ngnd,nvdd]),
  remove_p(I,nvdd,Q),
  remove_p(P,Q,O),
  remove_n(N,O,R),
  remove_n(I,R,ngnd),
  asserta(invZ(P,N,I,O,X,Y,1)),
  fail.
invZ.

/* Level-2 Prolog Extraction Rules */

nand :-
  ntrans(A,ngnd,Q,X,Y),
  ntrans(B,Q,O,_,_),
  not_connected([Q,O,nvdd,ngnd]),
  ptrans(A,nvdd,O,_,_),
  ptrans(B,nvdd,O,_,_),
  not_connected([Q,A]),
  not_connected([Q,B]),
  remove_n(A,ngnd,Q),
  remove_n(B,Q,O),
  remove_p(A,nvdd,O),
  remove_p(B,nvdd,O),
  find_anomaly_list(nand([A,B],O,X,Y,1),[Q]),
  asserta(nand([A,B],O,X,Y,1)),
  fail.
nand :-
  ntrans(A,ngnd,Q,X,Y),
  ptrans(A,nvdd,O,_,_),
  ptrans(B,nvdd,O,_,_),
  ntrans(B,Q,R,_,_),
  not_connected([Q,R,O,nvdd,ngnd]),
  ntrans(C,R,O,_,_),
  ptrans(C,nvdd,O,_,_),
  not_connected([Q,R,A]),
  not_connected([Q,R,B]),
    not_connected([Q,R,C]),
    remove_n(A,ngnd,Q),
    remove_n(B,Q,R),
    remove_n(C,R,O),
    remove_p(A,nvdd,O),
    remove_p(B,nvdd,O),
    remove_p(C,nvdd,O),
    find_anomaly_list(nand([A,B,C],O,X,Y,1),[Q,R]),
    asserta(nand([A,B,C],O,X,Y,1)),
    fail.
  nand.

  nor :-
    ptrans(A,nvdd,Q,X,Y),
    ptrans(B,Q,O,_,_),
    ntrans(A,ngnd,O,_,_),
    ntrans(B,ngnd,O,_,_),
    not_connected([Q,O,nvdd,ngnd]),
    not_connected([Q,A]),
    not_connected([Q,B]),
    remove_p(A,nvdd,Q),
    remove_p(B,Q,O),
    remove_n(A,ngnd,O),
    remove_n(B,ngnd,O),
    find_anomaly_list(nor([A,B],O,X,Y,1),[Q]),
    asserta(nor([A,B],O,X,Y,1)),
    fail.
  nor.

  /* Level-N Prolog Extraction Rules */

  dff :-
    tgate(P1,P2,D,Q,_,_),
    invZ(P2,P1,G,Q,X,Y,1),
    inv(Q,G,_,_,1),
    remove_tgate(P1,P2,D,Q),
    retract(invZ(P2,P1,G,Q,X,Y,1)),
    retract(inv(Q,G,_,_,1)),
    find_anomaly_list(dff(P1,P2,D,G,X,Y),[Q]),
    asserta(dff(P1,P2,D,G,X,Y,1)),
    fail.
  dff.
```

A discussion of these Prolog rules may be found in Section IV.

6.4 Prolog Error Identification Rules

Listed within this section are the Prolog error identification rules.

```
/* Prolog rules for finding errors in the transistor netlist */

find_error :-
  ptrans(G,nvdd,ngnd,X,Y),
  write('Bad trans, '),write(ptrans(G,nvdd,ngnd,X,Y)),
  write(': removed'),nl,
  remove_p(G,nvdd,ngnd),
  fail.
find_error :-
  ptrans(ngnd,nvdd,S,X,Y),
  write('Straight wire, '),write(ptrans(ngnd,nvdd,S,X,Y)),
  write(': removed'),nl,
  remove_p(ngnd,nvdd,S),
  fail.
find_error :-
  ptrans(nvdd,A,B,X,Y),
  write('Open connection, '),write(ptrans(nvdd,A,B,X,Y)),
  write(': removed'),nl,
  remove_p(nvdd,A,B),
  fail.
find_error :-
  ptrans(G,A,A,X,Y),
  write('Capacitor, '),write(ptrans(G,A,A,X,Y)),
  write(': removed'),nl,
  remove_p(G,A,A),
  fail.
find_error :-
  ntrans(G,nvdd,ngnd,X,Y),
  write('Bad trans, '),write(ntrans(G,nvdd,ngnd,X,Y)),
  write(': removed'),nl,
  remove_n(G,nvdd,ngnd),
  fail.
find_error :-
  ntrans(nvdd,ngnd,S,X,Y),
  write('Straight wire, '),write(ntrans(nvdd,ngnd,S,X,Y)),
  write(': removed'),nl,
  remove_n(nvdd,ngnd,S),
  fail.
find_error :-
  ntrans(ngnd,A,B,X,Y),
  write('Open connection, '),write(ntrans(ngnd,A,B,X,Y)),
  write(': removed'),nl,
  remove_n(ngnd,A,B),
  fail.
find_error :-
  ntrans(G,A,A,X,Y),
  write('Capacitor, '),write(ntrans(G,A,A,X,Y)),
  write(': removed'),nl,
  remove_n(G,A,A),
  fail.
find_error.
```

```
/* Prolog Error Identification Rules for Level-1 Components */

find_more_errors :-
  invZ(P,P,In,Out,X,Y,1),
  write('Screwy invZ, '),write(invZ(P,P,In,Out,X,Y,1)),
  write(': removed'),nl,
  retract(invZ(P,P,In,Out,X,Y,1)),
  fail.
find_more_errors :-
  invZ(Pg,Ng,Bad,Bad,X,Y,1),
  write('Oscillating invZ, '),write(invZ(Pg,Ng,Bad,Bad,X,Y,1)),
  write(': removed'),nl,
  retract(invZ(Pg,Ng,Bad,Bad,X,Y,1)),
  fail.
find_more_errors :-
  tgate(P,P,In,Out,X,Y,1),
  write('Screwy tgate, '),write(tgate(P,P,In,Out,X,Y,1)),
  write(': removed'),nl,
  retract(tgate(P,P,In,Out,X,Y,1)),
  fail.
find_more_errors :-
  tgate(Pg,Ng,Bad,Bad,X,Y,1),
  write('Worthless tgate, '),write(tgate(Pg,Ng,Bad,Bad,X,Y,1)),
  write(': removed'),nl,
  retract(tgate(Pg,Ng,Bad,Bad,X,Y,1)),
  fail.
find_more_errors :-
  inv(Bad,Bad,X,Y,1),
  write('Oscillating inv, '),write(inv(Bad,Bad,X,Y,1)),
  write(': removed'),nl,
  retract(inv(Bad,Bad,X,Y,1)),
  fail.
find_more_errors.

/* Prolog Rules for finding internal design errors */

find_anomaly_list(_,[]) :- !.
find_anomaly_list(Comp,[Node|Rest]) :-
  find_anomaly(Comp,Node),!,
  find_anomaly_list(Comp,Rest).
find_anomaly_list(Comp,[_|Rest]) :-
  find_anomaly_list(Comp,Rest).

find_anomaly(Comp,Node) :-
  (ptrans(Node,_,_,X,Y);
   ptrans(_,Node,_,X,Y);
   ntrans(Node,_,_,X,Y);
   ntrans(_,Node,_,X,Y)),
  write('Failure extracting component '),write(Comp),
  write('.'),nl,write('     Internal node, '),write(Node),
  write(', connected to a transistor at X:'),
  write(X),write(', Y:'),write(Y),write('.'),nl.
find_anomaly(Comp,Node) :-
  (inv(Node,_,X,Y,_);
   inv(_,Node,X,Y,_)),
  write('Failure extracting component '),write(Comp),
  write('.'),nl,write('     Internal node, '),write(Node),
  write(', connected to an inverter at X:'),
  write(X),write(', Y:'),write(Y),write('.'),nl.
find_anomaly(Comp,Node) :-
  (tgate(Node,_,_,_,X,Y,_);
   tgate(_,Node,_,_,X,Y,_);
   tgate(_,_,Node,_,X,Y,_);
   tgate(_,_,_,Node,X,Y,_)),
  write('Failure extracting component '),write(Comp),
  write('.'),nl,write('     Internal node, '),write(Node),
  write(', connected to a tgate at X:'),
  write(X),write(', Y:'),write(Y),write('.'),nl.
find_anomaly(Comp,Node) :-
  (invZ(Node,_,_,_,X,Y,_);
   invZ(_,Node,_,_,X,Y,_);
   invZ(_,_,Node,_,X,Y,_);
   invZ(_,_,_,Node,X,Y,_)),
  write('Failure extracting component '),write(Comp),
  write('.'),nl,write('     Internal node, '),write(Node),
  write(', connected to a high-impedance inverter at X:'),
  write(X),write(', Y:'),write(Y),write('.'),nl.
```

A discussion of these rules may be found in Section V.

6.5 Prolog Output Rules

The Prolog rules listed in this section are used to output the component netlist. They are normally used when attempting to use GES in a noninteractive mode.

```
list_trans :-
  retract(p(A,B,C,W,L,X,Y)),
  write(p(A,B,C,W,L,X,Y)),write('.'),nl,
  retract(n(A1,B1,C1,W1,L1,X1,Y1)),
  write(n(A1,B1,C1,W1,L1,X1,Y1)),write('.'),nl,!,
  list_trans.
list_trans :-
  retract(p(A,B,C,W,L,X,Y)),
  write(p(A,B,C,W,L,X,Y)),write('.'),nl,!,
  list_trans.
list_trans :-
  retract(n(A,B,C,W,L,X,Y)),
  write(n(A,B,C,W,L,X,Y)),write('.'),nl,!,
  list_trans.
list_trans.

list_invZ :-
  retract(invZ(A,B,C,D,E,F,G)),
  write(invZ(A,B,C,D,E,F,G)),write('.'),nl,!,
  list_invZ.
list_invZ.

list_inv :-
  retract(inv(A,B,C,D,E)),
  write(inv(A,B,C,D,E)),write('.'),nl,!,
  list_inv.
list_inv.

list_tgate :-
  retract(tgate(A,B,C,D,E,F,G)),
  write(tgate(A,B,C,D,E,F,G)),write('.'),nl,!,
  list_tgate.
list_tgate.

list_nand :-
  retract(nand(A,B,X,Y,T)),
  write(nand(A,B,X,Y,T)),write('.'),nl,!,
  list_nand.
list_nand.

list_nor :-
  retract(nor(A,B,C,D,E)),
  write(nor(A,B,C,D,E)),write('.'),nl,!,
  list_nor.
list_nor.

list_dff :-
  retract(dff(A,B,C,D,E,F,G)),
  write(dff(A,B,C,D,E,F,G)),write('.'),nl,!,
  list_dff.
list_dff.
```

A discussion of these rules may be found in Section IV.

6.6 Prolog Main Driver

The main driver for a GES system is shown in this section. Additionally, a Quintus Prolog command directive is shown. The directive

```
:- unknown('trace','fail').
```

is used to tell the Prolog interpreter not to go into trace mode is an unknown Prolog rule or fact is encountered during execution. This directive is necessary only if using Quintus Prolog.

```
:- unknown('trace','fail').

ges :-
  see('good.pro'),
  remove_dup_trans,
  seen,
  write('finished with read.'),nl,
  find_error,
  write('finished with find_error.'),nl,
  inv,
  write('finished inverters.'),nl,
  tgate,
  write('finished tgates.'),nl,
  invZ,
  write('finished invZ.'),nl,
  nand,
  write('finished nand.'),nl,
  nor,
  write('finished nor.'),nl,
  find_more_errors,
  write('finished find_more_errors.'),nl,
  dff,
  write('finished dff.'),nl,
  tell(outcomp),
  list_inv,
  list_invZ,
  list_tgate,
  list_nand,
  list_nor,
  list_dff,
  told,
  tell(outtrans),
  list_trans,
  told,
  halt.
```

There is one input file required by the main driver. Two output files are also produced. The file `good.pro` is the input transistor netlist. The files **`outtrans`** and **`outcomp`** are used to collect the leftover transistors and extracted components, respectively.

With the information presented in this section, a modest GES system for extraction and identification of errors may be constructed. The section on examples demonstrates GES being used on a custom VLSI layout design for a clock generator.

VII. Generating VHDL and HOL From Component Netlists

Contained in this section is a discussion of the Prolog code used to generate VHDL and HOL. The code is template-based; it generates VHDL or HOL by filling in the blanks in the templates using information from the components in the Prolog facts database. First, the required input files will be listed. Next, some commonalities between VHDL and HOL will be discussed. Afterwards, a discussion of the Prolog code for generating VHDL will be provided. Finally, the changes to the VHDL generation code required to generate HOL will be enumerated.

7.1 Requirements

In order to generate VHDL and HOL code the following files are required.

component	Contains the component name and lists of inputs and outputs.
outcomp	Contains a netlist of components above the transistor level.
outtrans	Contains a transistor netlist.

The two files, **outcomp** and **outtrans**, have been explained previously. The file, **component**, is explained below.

The file, **component**, contains one record. The record is a Prolog fact of the following format.

component(*comp_name*,[*comp_In*],[*comp_Out*], [*comp_InOut*],[*comp_Buf*],[*comp_Link*]).

where

comp_name	The name of the component being extracted.
comp_In	The list of signals of mode in.
comp_Out	The list of signals of mode out.
comp_InOut	The list of signals of mode inout.
comp_Buf	The list of signals of mode buffer.
comp_Link	The list of signals of mode linkage.

The term "mode" is taken from the *VHDL Language Reference Manual* [3].

7.2 Generating Hardware Descriptions

VHDL and HOL share some common language constructs for describing hardware. These constructs are

- name of the component being described,
- interface list of the component being described,
- an area for declaring signals internal to the component being described,
- and an area for enumerating the internal parts of the component being described.

VHDL has one additional requirement not found within a HOL description. The internal components must be declared in the architecture declarative area prior to their instantiation within the architecture body. As it turns out, this additional requirement is not difficult to fulfill.

7.3 Generating VHDL

The following is a description of the Prolog code used to generate VHDL descriptions. The generator is divided into three main parts. The first part reads in the files **component**, **outcomp**, and **outtrans**. The second part generates the VHDL entity part, signal declarations, and component declarations for the component. The final part generates the instantiated subcomponents in the VHDL architecture body.

The main driver of the VHDL generator is the following.

```
pro2vhdl :-
  read_in_files,
  write('Finished with read'),nl,
  tell('outfile.vhd'),
  pass1,
  pass2,
  told,
  halt.
```

The three main parts of the generator are called in sequence.

The first part, **read_in_files/0**, is shown below.

```
read_in_files :-
  see('component'),
  read(X),
  get_net(X),
  seen,
  see('outcomp'),
  read(Y),
  get_net(Y),
  seen,
  see('outtrans'),
  read(Z),
  get_net(Z),
  seen.
```

see/1 and **read/1** are standard Prolog I/O routines for opening a file and reading from the opened file. The Prolog function, **get_net/1**, reads in the opened file, checking for **end_of_file** and asserting the input onto the Prolog fact database. **get_net/1** is shown below.

```
get_net(end_of_file) :- !.
get_net(X) :-
  assert(X),
  read(Y),!,
  get_net(Y).
```

Once the three files, **component**, **outcomp**, and **outtrans**, have been read in, the Prolog function, **pass1/0**, is called. **pass1/0** is shown below.

```
pass1 :-
  gen_header_1,
  gen_sig,
  gen_header_2a,
  gen_header_3.
```

The Prolog function, **pass1/0**, then calls four Prolog functions, **gen_header_1/0**, **gen_sig/0**, **gen_header_2a/0**, and **gen_header_3/0**. **gen_header_1/0** generates the VHDL entity declaration for the component. **gen_header_1/0** is shown below.

```
gen_header_1 :-
  component(Name,ModeIn,ModeOut,ModeInOut,ModeBuf,ModeLink),
  write('entity '),write(Name),write(' is'),nl,
  gen_port_list(ModeIn,ModeOut,ModeInOut,ModeBuf,ModeLink),
  write('end '),write(Name),write(';'),nl,
  write('architecture structural of '),write(Name),write(' is'),nl.
```

gen_header_1/0 uses the **component** Prolog fact to generate the entity identifier and the port clause of the entity declaration. **gen_port_list/5** generates the port clause ensuring semicolons are placed properly. **gen_port_list/5** is shown below.

```
gen_port_list([],[],[],[],[]) :- !.
gen_port_list(ModeIn,ModeOut,ModeInOut,ModeBuf,ModeLink) :-
  write('    port('),nl,
  gen_port_signal(ModeIn,in),
  gen_semicolon1(ModeOut),
  gen_port_signal(ModeOut,out),
  gen_semicolon1(ModeInOut),
  gen_port_signal(ModeInOut,inout),
  gen_semicolon1(ModeBuf),
  gen_port_signal(ModeBuf,buffer),
  gen_semicolon1(ModeLink),
  gen_port_signal(ModeLink,linkage),
  write('        );'),nl.
```

gen_port_list/5 uses two subordinate Prolog functions, **gen_port_signal/2** and **ges_semicolon1/1**. **gen_port_signal/2** generates the signal declarations within the port clause using the mode information passed with the signal list. **gen_semicolon1/1** generates semicolons conforming to the syntax for a port clause. **gen_port_signal/2** is shown below.

```
gen_port_signal([],_) :- !.
gen_port_signal([Head|Tail],Mode) :-
  semicolon,
  gen_semicolon([Head|Tail]),
  asserta(not_signal(Head)),
  write('        signal '),write(Head),write(' : '),write(Mode),
  write(' mos_node'),
  !,
  gen_port_signal(Tail,Mode).
gen_port_signal([Head|Tail],Mode) :-
  assert(semicolon),
  asserta(not_signal(Head)),
  write('        signal '),write(Head),write(' : '),write(Mode),
  write(' mos_node'),
  !,
  gen_port_signal(Tail,Mode).
```

An additional "helper" Prolog function, **gen_semicolon/1**, is also used to generate semicolons in the correct position. Both **gen_semicolon1/1** and **gen_semicolon/1** are shown below.

```
gen_semicolon([]) :- !.
gen_semicolon(_) :-
  write(';'),nl,!.

gen_semicolon1([]) :- !.
gen_semicolon1(_) :-
  retract(semicolon),
  write(';'),nl,!.
gen_semicolon1(_) :- !.
```

Some typical entity declarations are shown for various **component** Prolog facts. For a **component** of the following form,

```
component(clkgen,[],[],[],[],[]).
component(clkgen,[nIZ_GO,nIZ_CAP1,nIZ_CAP2],[nOZ_PQ1,nOZ_PQ2],[],[],[]).
component(cordic,[],[nRESULT],[nA,nB],[],[]).
component(complement,[nIN],[],[nOUT],[],[nvdd,ngnd]).
```

the following port clauses would result:

```
entity clkgen is
end clkgen;


entity clkgen is
    port(
        signal nIZ_GO : in mos_node;
        signal nIZ_CAP1 : in mos_node;
        signal nIZ_CAP2 : in mos_node;
        signal nOZ_PQ1 : out mos_node;
        signal nOZ_PQ2 : out mos_node        );
end clkgen;


entity cordic is
    port(
        signal nRESULT : out mos_node;
        signal nA : inout mos_node;
        signal nB : inout mos_node        );
end cordic;


entity complement is
    port(
        signal nIN : in mos_node;
        signal nOUT : inout mos_node;
        signal nvdd : linkage mos_node;
        signal ngnd : linkage mos_node        );
end complement;
```

The Prolog function, **gen_sig/0**, compiles a list of signals from the component netlist and generates the appropriate signal assignment statements in the architecture declarative part. **gen_sig/0** ensures that there are no identical signal assignment statements and that no signal is declared in the architecture declarative part that was already declared in the port clause of the entity declaration. The **gen_sig/0** Prolog function is shown below.

```
gen_sig :-
  !,
  gen_sig_inv,
  gen_sig_ptrans,
  gen_sig_ntrans,
  gen_sig_nand,
  gen_sig_nor,
  gen_sig_invZ,
  gen_sig_tgate,
  gen_sig_state.
```

The Prolog functions **gen_sig_inv/0** through **gen_sig_tgate/0** perform the same basic function. **gen_sig_inv/0** through **gen_sig_tgate/0** are shown below.

```
gen_sig_invZ :-
  retract(invZ(A,B,C,D,X,Y,1)),
  add_signal(A),
  add_signal(B),
  add_signal(C),
  add_signal(D),!,
  gen_sig_invZ,
  asserta(invZ(A,B,C,D,X,Y,1)).
gen_sig_invZ.

gen_sig_tgate :-
  retract(tgate(A,B,C,D,X,Y,1)),
  add_signal(A),
  add_signal(B),
  add_signal(C),
  add_signal(D),!,
  gen_sig_tgate,
  asserta(tgate(A,B,C,D,X,Y,1)).
gen_sig_tgate.

gen_sig_ptrans :-
  retract(p(A,B,C,L,W,X,Y)),
  add_signal(A),
  add_signal(B),
  add_signal(C),!,
  gen_sig_ptrans,
  asserta(p(A,B,C,L,W,X,Y)).
gen_sig_ptrans.

gen_sig_ntrans :-
  retract(n(A,B,C,L,W,X,Y)),
  add_signal(A),
  add_signal(B),
  add_signal(C),!,
  gen_sig_ntrans,
  asserta(n(A,B,C,L,W,X,Y)).
gen_sig_ntrans.

gen_sig_inv :-
  retract(inv(A,B,X,Y,1)),
  add_signal(A),
  add_signal(B),!,
  gen_sig_inv,
  asserta(inv(A,B,X,Y,1)).
gen_sig_inv.

gen_sig_nand :-
  retract(nand([A|B],C,X,Y,1)),
  add_signal(A),
  add_signal(C),
  gen_sig_nand_rem(B),!,
  gen_sig_nand,
  asserta(nand([A|B],C,X,Y,1)).
gen_sig_nand.
```

```
gen_sig_nand_rem([A|B]) :-
  add_signal(A),!,
  gen_sig_nand_rem(B).
gen_sig_nand_rem([]).

gen_sig_nor :-
  retract(nor([A|B],C,X,Y,1)),
  add_signal(A),
  add_signal(C),
  gen_sig_nand_rem(B),!,
  gen_sig_nor,
  asserta(nor([A|B],C,X,Y,1)).
gen_sig_nor.
```

The **add_signal/1** Prolog function adds a signal name only if it doesn't already exist in the Prolog facts database. **add_signal/1** is shown below.

```
add_signal(A) :-
  signal(A),!.
add_signal(A) :-
  not_signal(A),!.
add_signal(A) :-
  asserta(signal(A)),!.
```

The Prolog function, **gen_sig_state/0**, generates the signal assignment statements for the signals contained in the Prolog facts database. **gen_sig_state/0** is shown below.

```
gen_sig_state :-
  retract(signal(X)),
  write('       signal '),
  write(X),write(' : mos_node;'),nl,!,
  gen_sig_state.
gen_sig_state.
```

The Prolog function **gen_header_2a/0** calls the Prolog functions **gen_head_inv/0** through **gen_head_tgate/0** for generating the component declarations for the components to be instantiated in the architecture body. The component facts database is checked for the existence of a a particular component before generating a component declaration. That way component declarations are not made when the related component does not exist in the Prolog facts database. The Prolog functions, **gen_head_inv/0** through **gen_head_tgate/0**, are shown below.

```
gen_head_invZ :-
  invZ(_,_,_,_,_,1),
  write('       component invZ'),nl,
  write('                 generic ( constant tPLH: TIME := 0 ns;'),nl,
  write('                     constant tPHL: TIME := 0 ns);'),nl,
  write('                 port ( vdd  : inout mos_node;'),nl,
  write('                        gnd  : inout mos_node;'),nl,
  write('                        p1   : in mos_node;'),nl,
  write('                        p2   : in mos_node;'),nl,
  write('                        g    : in mos_node;'),nl,
  write('                        d    : inout mos_node);'),nl,
  write('                 end component;'),nl,
  write('    for all : invZ use entity work.invZ (invZ);'),nl.
gen_head_invZ.

gen_head_tgate :-
  tgate(_,_,_,_,_,_,1),
  write('          component tgate'),nl,
  write('                 generic ( constant tPLH: TIME := 0 ns;'),nl,
  write('                     constant tPHL: TIME := 0 ns);'),nl,
  write('                 port ( p1 : in mos_node;'),nl,
  write('                        p2 : in mos_node;'),nl,
  write('                        g  : inout mos_node;'),nl,
  write('                        d  : inout mos_node);'),nl,
  write('                 end component;'),nl,
  write('    for all : tgate use entity work.tgate(tgate);'),nl.
gen_head_tgate.

gen_head_ptrans :-
  p(_,_,_,_,_,_,_),
  write('          component ptrans'),nl,
  write('                 generic (constant gate_length: integer;'),nl,
  write('                     constant gate_width: integer);'),nl,
  write('                   port ( Gate    : in mos_node;'),nl,
  write('                          Drain   : inout mos_node;'),nl,
  write('                          Source  : inout mos_node);'),nl,
  write('                 end component;'),nl,
  write('       for all : ptrans use entity work.ptrans( ptrans );'),nl.
gen_head_ptrans.

gen_head_ntrans :-
  n(_,_,_,_,_,_,_),
  write('          component ntrans'),nl,
  write('                 generic (constant gate_length: integer;'),nl,
  write('                     constant gate_width: integer);'),nl,
  write('                   port ( Gate    : in mos_node;'),nl,
  write('                          Drain   : inout mos_node;'),nl,
  write('                          Source  : inout mos_node);'),nl,
  write('                 end component;'),nl,
  write('       for all : ntrans use entity work.ntrans( ntrans );'),nl.
gen_head_ntrans.

gen_head_inv :-
  inv(_,_,_,_,1),
  write('            component INV'),nl,
  write('                   generic ( constant tPLH: TIME := 0 ns;'),nl,
  write('                       constant tPHL: TIME := 0 ns);'),nl,
  write('                   port ( signal A: in mos_node;'),nl,
  write('                   signal B: out mos_node);'),nl,
  write('               end component;'),nl,
  write('         for all : inv use entity work.inv( inv );'),nl.
gen_head_inv.

gen_head_nand :-
  nand(_,_,_,_,1),
  write('            component NAND_GATE'),nl,
  write('                   generic ( constant tPLH: TIME := 0 ns;'),nl,
  write('                       constant tPHL: TIME := 0 ns);'),nl,
  write('                   port ( signal A: in mos_node;'),nl,
  write('                   signal B: in mos_node;'),nl,
  write('                   signal C: out mos_node); ),nl,
  write('               end component;'),nl,
  write('         for all : nand_gate use entity work.nand_gate( nand_gate );'),
  nl.
gen_head_nand.

gen_head_mdffnr :-
  dff(_,_,_,_,_,_,1),
  write('            component DFFNR'),nl,
  write('                port ( signal D: in mos_node;'),nl,
  write('                signal Q: inout mos_node;'),nl,
  write('                signal PQ: in mos_node;'),nl,
  write('                signal PQ_BAR: in mos_node);'),nl,
  write('                end component;'),nl,
  write('        for all : DFFNR use entity work.dffnr(DFFNR);'),nl.
gen_head_mdffnr.
```

The Prolog function **gen_header_3/0** generates the VHDL code that designates the beginning of the architecture body and generates code for a VHDL process. The VHDL process is left empty for the user to place model-specific code. **gen_header_3/0** is shown below.

```
gen_header_3 :-
  write('begin'),nl,
  write('process'),nl,
  write('    begin'),nl,
  write('    wait;'),nl,
  write('    end process;'),nl.
```

The final part of the VHDL generator, **pass2/0**, is now discussed. The final part generates an instantiated component for its respective component in the component netlist. **pass2/0** is shown below.

```
pass2 :-
  gen_body,
  gen_tail.
```

The Prolog function **gen_body/0** calls the Prolog functions that produce the instantiated components. A counter variable, X, is passed to each subfunction of **gen_body/0** to be used as a means of generating unique identifiers for each instantiated component in the architecture body. **gen_body/0** is shown below.

```
gen_body :-
  X is 0,
  ptrans_port(X),
  ntrans_port(X),
  inv_port(X),
  nand_port(X),
  nor_port(X),
  dffnr_port(X),
  invZ_port(X),
  tgate_port(X).
```

The subordinate Prolog functions of **gen_body/0** are templates for generating each component instantiation. The template-type Prolog functions are listed below.

```
invZ_port(Y) :-
  X is Y + 1,
  retract(invZ(A,B,C,D,Xloc,Yloc,1)),
  write('       invZ'),write(X),
  write(':invZ -- at X:'),write(Xloc),write(',Y:'),write(Yloc),nl,
  write('                 generic map( tPLH => 0 ns,'),nl,
  write('                     tPHL => 0 ns)'),nl,
  write('       port map ( vdd => vdd,'),nl,
  write('                  gnd => gnd,'),nl,
  write('                  p1  =>'),write(A),write(','),nl,
  write('                  p2  =>'),write(B),write(','),nl,
  write('                  g   =>'),write(C),write(','),nl,
  write('                  d   =>'),write(D),write(');'),nl,!,
  invZ_port(X).
invZ_port(_).

tgate_port(Y) :-
  X is Y + 1,
  retract(tgate(A,B,C,D,Xloc,Yloc,1)),
  write('    TGATE'),write(X),
  write(':TGATE -- at X:'),write(Xloc),write(',Y:'),write(Yloc),nl,
  write('                 generic map( tPLH => 0 ns,'),nl,
  write('                     tPHL => 0 ns)'),nl,
  write('    port map( p1 =>'),write(A),write(','),nl,
  write('              p2 =>'),write(B),write(','),nl,
  write('              g  =>'),write(C),write(','),nl,
  write('              d  =>'),write(D),write(');'),nl,!,
  tgate_port(X).
tgate_port(_).

ntrans_port(Y) :-
  X is Y+1,
  retract(n(A,B,C,L,W,Xloc,Yloc)),
  write('    NTRANS'),write(X),
  write(':NTRANS -- at X:'),write(Xloc),write(',Y:'),write(Yloc),nl,
  write('           generic map(gate_length => '),write(L),write(','),nl,
  write('                  gate_width => '),write(W),write(')'),nl,
  write('           port map( Gate =>'),write(A),write(','),nl,
  write('                    Drain =>'),write(B),write(','),nl,
  write('                   Source =>'),write(C),write(');'),nl,!,
  ntrans_port(X).
ntrans_port(_).

ptrans_port(Y) :-
  X is Y+1,
  retract(p(A,B,C,L,W,Xloc,Yloc)),
  write('    PTRANS'),write(X),
  write(':PTRANS -- at X:'),write(Xloc),write(',Y:'),write(Yloc),nl,
  write('           generic map(gate_length => '),write(L),write(','),nl,
  write('                  gate_width => '),write(W),write(')'),nl,
  write('           port map( Gate =>'),write(A),write(','),nl,
  write('                    Drain =>'),write(B),write(','),nl,
  write('                   Source =>'),write(C),write(');'),nl,!,
  ptrans_port(X).
ptrans_port(_).

inv_port(Y) :-
  X is Y+1,
  retract(inv(A,B,Xloc,Yloc,1)),
  write('            INV'),write(X),
  write(':INV -- at X:'),write(Xloc),write(',Y:'),write(Yloc),nl,
  write('               generic map( tPLH => 0 ns,'),rl,
  write('                  tPHL => 0 ns)'),nl,
  write('               port map( A =>'),write(A),write(','),nl,
  write('                  B =>'),write(B),write(');'),nl,!,
  inv_port(X).
inv_port(_).

nand_port(Y) :-
  X is Y + 1,
  retract(nand([A,B],C,Xloc,Yloc,1)),
  write('            NAND_GATE'),write(X),
  write(': NAND_GATE -- at X:'),write(Xloc),write(',Y:'),write(Yloc),nl,
  write('               generic map( tPLH => 0 ns,'),nl,
  write('                     tPHL => 0 ns)'),nl,
  write('                port map( A => '),write(A),write(','),nl,
  write('                     B => '),write(B),write(','),nl,
  write('                     C => '),write(C),write(');'),nl,!,
  nand_port(X).
nand_port(_).

nor_port(Y) :-
  X is Y + 1,
  retract(nor([A,B],C,Xloc,Yloc,1)),
  write('           NOR_GATE'),write(X),
  write(': NOR_GATE -- at X:'),write(Xloc),write(',Y:'),write(Yloc),nl,
  write('                generic map( tPLH => 0 ns,'),nl,
  write('                     tPHL => 0 ns)'),nl,
  write('                port map( A => '),write(A),write(','),nl,
  write('                     B => '),write(B),write(','),nl,
  write('                     C => '),write(C),write(');'),nl,!,
  nor_port(X).
nor_port(_).

dffnr_port(Y) :-
  X is Y + 1,
  retract(dff(A,B,C,D,Xloc,Yloc,1)),
  write('       DFFNR'),write(X),
  write(':DFFNR -- at X:'),write(Xloc),write(',Y:'),write(Yloc),nl,
  write('                   port map(D =>'),write(C),write(','),nl,
  write('                            Q =>'),write(D),write(','),nl,
  write('                           PQ =>'),write(B),write(','),nl,
  write('                       PQ_BAR =>'),write(A),write(');'),nl,!,
  dffnr_port(X).
dffnr_port(_).
```

The final Prolog function called under **pass3/0** is **gen_tail/0**. Its purpose is to produce the final line of VHDL code that terminates the architecture body. **gen_tail/0** is shown below.

```
gen_tail :-
  write('end structural;'),nl.
```

The Prolog code discussed in this section is sufficient for generating VHDL descriptions from extracted component netlists using the GES system described in Section VI. To add additional Prolog functions for additional component types, the following must be modified: **gen_header_2a/0**, **gen_sig/0**, and **gen_body/0**.

7.4 Generating HOL

Many of the routines discussed in the section on generating VHDL are also used in the Prolog code for generating HOL [8,9]. The component description generated from this system is actually a ML [10] definition describing the component. The only real difference between the VHDL generation and HOL generation is in the format of the output. To demonstrate the similarity, we will generate VHDL and HOL from the files **component**, **outcomp**, and **outtrans**.

The file **component** contains the following.

```
component(clkgen,[nIZ_GO,nIZ_CAP1,nIZ_CAP2],[nOZ_PQ1,nOZ_PQ2],[],[],[]).
```

The file **outcomp** contains the following.

```
nand([n3_10_21,n3_44_29],n3_40_12,18,-9,1).
nor([n3_238_103,n3_226_12],n3_72_106,141,8,1).
nor([n3_72_106,n3_330_10],n3_238_103,155,8,1).
```

The file **outtrans** is empty.

The following is the VHDL code produced from the files **component**, **outcomp**, and **outtrans**.

```
entity clkgen is
    port(
        signal nIZ_GO : in mos_node;
        signal nIZ_CAP1 : in mos_node;
        signal nIZ_CAP2 : in mos_node;
        signal nOZ_PQ1 : out mos_node;
        signal nOZ_PQ2 : out mos_node        );
end clkgen;
architecture structural of clkgen is
        signal n3_330_10 : mos_node;
        signal n3_226_12 : mos_node:
        signal n3_72_106 : mos_node;
        signal n3_238_103 : mos_node;
        signal n3_44_29 : mos_node;
        signal n3_40_12 : mos_node;
        signal n3_10_21 : mos_node;
            component NAND_GATE
                    generic ( constant tPLH: TIME := 0 ns;
                        constant tPHL: TIME := 0 ns);
                    port ( signal A: in mos_node;
                    signal B: in mos_node;
                    signal C: out mos_node);
                end component;
         for all : nand_gate use entity work.nand_gate( nand_gate );
            component NOR_GATE
                    generic ( constant tPLH: TIME := 0 ns;
                        constant tPHL: TIME := 0 ns);
                    port ( signal A: in mos_node;
                    signal B: in mos_node;
                    signal C: out mos_node);
                end component;
            for all : nor_gate use entity work.nor_gate( nor_gate );
begin
process
    begin
    wait;
    end process;
            NAND_GATE1: NAND_GATE -- at X:18,Y:-9
                generic map( tPLH => 0 ns,
                    tPHL => 0 ns)
                port map( A => n3_10_21,
                    B => n3_44_29,
                    C => n3_40_12);
            NOR_GATE1: NOR_GATE -- at X:141,Y:8
                generic map( tPLH => 0 ns,
                    tPHL => 0 ns)
                port map( A => n3_10_21,
                    B => n3_44_29,
                    C => n3_40_12);
            NOR_GATE1: NOR_GATE -- at X:141,Y:8
                generic map( tPLH => 0 ns,
                    tPHL => 0 ns)
                port map( A => n3_238_103,
                    B => n3_226_12,
                    C => n3_72_106);
            NOR_GATE2: NOR_GATE -- at X:155,Y:8
                generic map( tPLH => 0 ns,
                    tPHL => 0 ns)
                port map( A => n3_72_106,
                    B => n3_330_10,
                    C => n3_238_103);
end structural;
```

The following is the ML definition produced from the files **component**, **outcomp**, and **outtrans**. The ML definition may then be loaded into HOL for formal verification.

```
let clkgen =
  new_definition('clkgen',
    "!
      (nIZ_GO:bool)
      (nIZ_CAP1:bool)
      (nIZ_CAP2:bool)
      (nOZ_PQ1:bool)
      (nOZ_PQ2:bool)
      .
      clkgen
        nIZ_GO
        nIZ_CAP1
        nIZ_CAP2
        nOZ_PQ1
        nOZ_PQ2
 =
(?
        n3_330_10
        n3_226_12
        n3_72_106
        n3_238_103
        n3_44_29
        n3_40_12
        n3_10_21
        .
    nand_gate n3_10_21 n3_44_29 n3_40_12  % at X:18,Y:-9%
    /\     nor_gate n3_238_103 n3_226_12 n3_72_106  % at X:141,Y:8%
    /\     nor_gate n3_72_106 n3_330_10 n3_238_103  % at X:155,Y:8%
)");;
```

From the ML definition above, the following is the same as the entity declaration in VHDL.

```
let clkgen =
  new_definition('clkgen',
    "!
      (nIZ_GO:bool)
      (nIZ_CAP1:bool)
      (nIZ_CAP2:bool)
      (nOZ_PQ1:bool)
      (nOZ_PQ2:bool)
      .
      clkgen
        nIZ_GO
        nIZ_CAP1
        nIZ_CAP2
        nOZ_PQ1
        nOZ_PQ2
 =
```

The part of the ML definition corresponding to the VHDL signal declarations in the architecture declarative part is shown below.

```
(?
        n3_330_10
        n3_226_12
        n3_72_106
        n3_238_103
        n3_44_29
        n3_40_12
        n3_10_21
```

The rest of the ML definition corresponds to the architecture body.

The Prolog code for HOL generation is presented with little discussion since only the templates differ from VHDL generation.

```
:- unknown(trace,fail).

pro2hol_version :-
  write('version 1.1'),nl.

pro2hol :-
  see('component'),
  read(X),
  get_net(X),
  seen,
  see('outcomp'),
  read(Y),
  get_net(Y),
  seen,
  see('outtrans'),
  read(Z),
  get_net(Z),
  seen,
  write('Finished with read'),nl,
  tell('outfile.ml'),
  pass1,
  pass2,
  told,
  halt.

get_net(end_of_file) :- !.
get_net(X) :-
  assert(X),
  read(Y),!,
  get_net(Y).


pass1 :-
  gen_header_1,
  gen_sig.

pass2 :-
  gen_body,
  gen_tail.


gen_header_1 :-
  component(Name,ModeIn,ModeOut,ModeInOut,ModeBuf,ModeLink),
  write('let '),write(Name),write(' ='),nl,
  write('  new_definition('''),write(Name),write(''','),nl,
  write('    "!'),nl,
  gen_port_signal(ModeIn),
  gen_port_signal(ModeOut),
  gen_port_signal(ModeInOut),
  gen_port_signal(ModeBuf),
  gen_port_signal(ModeLink),
  write('         .'),nl,
  write('       '),write(Name),nl,
  gen_port_signal2(ModeIn),
  gen_port_signal2(ModeOut),
  gen_port_signal2(ModeInOut),
  gen_port_signal2(ModeBuf),
  gen_port_signal2(ModeLink),
  write(' ='),nl.

gen_port_signal([]) :- !.
gen_port_signal([Head|Tail]) :-
  asserta(not_signal(Head)),
  write('       ('),write(Head),write(':bool)'),nl,!,
  gen_port_signal(Tail).

gen_port_signal2([]) :- !.
gen_port_signal2([Head|Tail]) :-
  write('         '),write(Head),nl,!,
  gen_port_signal2(Tail).


gen_sig :-
  !,gen_sig_inv,
  gen_sig_ptrans,
  gen_sig_ntrans,
  gen_sig_nand,
  gen_sig_nor,
  gen_sig_dffnr,
  gen_sig_invZ,
  gen_sig_tgate,
  write('(?'),nl,
  gen_sig_state,
  write('         .'),nl.

add_signal(A) :-
  signal(A),!.
add_signal(A) :-
  not_signal(A),!.
add_signal(A) :-
  asserta(signal(A)),!.


gen_sig_state :-
```

```
  retract(signal(X)),
  write('       '),write(X),nl,!,
  gen_sig_state.
gen_sig_state.

gen_sig_invZ :-
  retract(invZ(A,B,C,D,X,Y,1)),
  add_signal(A),
  add_signal(B),
  add_signal(C),
  add_signal(D),!,
  gen_sig_invZ,
  asserta(invZ(A,B,C,D,X,Y,1)).
gen_sig_invZ.

gen_sig_tgate :-
  retract(tgate(A,B,C,D,X,Y,1)),
  add_signal(A),
  add_signal(B),
  add_signal(C),
  add_signal(D),!,
  gen_sig_tgate,
  asserta(tgate(A,B,C,D,X,Y,1)).
gen_sig_tgate.

gen_sig_ptrans :-
  retract(p(A,B,C,L,W,X,Y)),
  add_signal(A),
  add_signal(B),
  add_signal(C),!,
  gen_sig_ptrans,
  asserta(p(A,B,C,L,W,X,Y)).
gen_sig_ptrans.

gen_sig_ntrans :-
  retract(n(A,B,C,L,W,X,Y)),
  add_signal(A),
  add_signal(B),
  add_signal(C),!,
  gen_sig_ntrans,
  asserta(n(A,B,C,L,W,X,Y)).
gen_sig_ntrans.

gen_sig_inv :-
  retract(inv(A,B,X,Y,1)),
  add_signal(A),
  add_signal(B),!,
  gen_sig_inv,
  asserta(inv(A,B,X,Y,1)).
gen_sig_inv.
```

```
gen_sig_nand :-
  retract(nand([A|B],C,X,Y,1)),
  add_signal(A),
  add_signal(C),
  gen_sig_nand_rem(B),!,
  gen_sig_nand,
  asserta(nand([A|B],C,X,Y,1)).
gen_sig_nand.
gen_sig_nand_rem([A|B]) :-
  add_signal(A),!,
  gen_sig_nand_rem(B).
gen_sig_nand_rem([]).

gen_sig_nor :-
  retract(nor([A|B],C,X,Y,1)),
  add_signal(A),
  add_signal(C),
  gen_sig_nand_rem(B),!,
  gen_sig_nor,
  asserta(nor([A|B],C,X,Y,1)).
gen_sig_nor.

gen_sig_dffnr:-
  retract(dff(A,B,C,D,X,Y,1)),
  add_signal(A),
  add_signal(B),
  add_signal(C),
  add_signal(D),!,
  gen_sig_dffnr,
  asserta(dff(A,B,C,D,X,Y,1)).
gen_sig_dffnr.


gen_body :-
  ptrans_port(Conj),
  ntrans_port(Conj),
  inv_port(Conj),
  nand_port(Conj),
  nor_port(Conj),
  dffnr_port(Conj),
  invZ_port(Conj),
  tgate_port(Conj).

conjunct(Conj) :- var(Conj),!.
conjunct(good) :-
  write('    /\ '),!.

invZ_port(Conj) :-
  retract(invZ(A,B,C,D,Xloc,Yloc,1)),
  conjunct(Conj),
  write('    invZ '),
  write(A),write(' '),write(B),write(' '),write(C),write(' '),
  write(D),
  write(' % at X:'),write(Xloc),write(',Y:'),write(Yloc),write('%'),nl,!,
  Conj = good,
  invZ_port(Conj).
invZ_port(_).

tgate_port(Conj) :-
  retract(tgate(A,B,C,D,Xloc,Yloc,1)),
  conjunct(Conj),
  write('    tgate '),
  write(A),write(' '),write(B),write(' '),write(C),write(' '),
  write(D),
  write(' % at X:'),write(Xloc),write(',Y:'),write(Yloc),write('%'),nl,!,
  Conj = good,
  tgate_port(Conj).
tgate_port(_).

ntrans_port(Conj) :-
  retract(n(A,B,C,_,_,Xloc,Yloc)),
  conjunct(Conj),
  write('    ntrans '),
  write(A),write(' '),write(B),write(' '),write(C),write(' '),
  write(' % at X:'),write(Xloc),write(',Y:'),write(Yloc),write('%'),nl,!,
  Conj = good,
  ntrans_port(Conj).
ntrans_port(_).

ptrans_port(Conj) :-
  retract(p(A,B,C,_,_,Xloc,Yloc)),
  conjunct(Conj),
  write('    ptrans '),
  write(A),write(' '),write(B),write(' '),write(C),write(' '),
  write(' % at X:'),write(Xloc),write(',Y:'),write(Yloc),write('%'),nl,!,
  Conj = good,
  ptrans_port(Conj).
ptrans_port(_).

inv_port(Conj) :-
  retract(inv(A,B,Xloc,Yloc,1)),
  conjunct(Conj),
  write('    inv '),
  write(A),write(' '),write(B),write(' '),
  write(' % at X:'),write(Xloc),write(',Y:'),write(Yloc),write('%'),nl,!,
  Conj = good,
  inv_port(Conj).
inv_port(_).

nand_port(Conj) :-
  retract(nand([A,B],C,Xloc,Yloc,1)),
  conjunct(Conj),
  write('    nand_gate ' ),
  write(A),write(' '),write(B),write(' '),write(C),write(' '),
  write(' % at X:'),write(Xloc),write(',Y:'),write(Yloc),write('%'),nl,!,
  Conj = good,
  nand_port(Conj).
nand_port(_).

nor_port(Conj) :-
  retract(nor([A,B],C,Xloc,Yloc,1)),
  conjunct(Conj),
  write('    nor_gate '),
  write(A),write(' '),write(B),write(' '),write(C),write(' '),
  write(' % at X:'),write(Xloc),write(',Y:'),write(Yloc),write('%'),nl,!,
  Conj = good,
  nor_port(Conj).
nor_port(_).

dffnr_port(Conj) :-
  retract(dff(A,B,C,D,Xloc,Yloc,1)),
  conjunct(Conj),
  write('    dffnr '),
  write(A),write(' '),write(B),write(' '),write(C),write(' '),
  write(D),
  write(' % at X:'),write(Xloc),write(',Y:'),write(Yloc),write('%'),nl,!,
  Conj = good,
  dffnr_port(Conj).
dffnr_port(_).

gen_tail :-
  write(')");;'),nl.
```

VIII. Example

In this section, two layout designs in *magic* will be examined. The first is a rather simple design of a fabricated two-phase clock generator. The second design is a large 60,000 transistor design. In both examples, GES was used in the layout process to identify external and internal design errors. In the first example, GES was used to generate a VHDL description of the clock generator for simulation of the design. The second example was used to demonstrate the performance of GES on a large custom VLSI chip design.

8.1 Clock Generator Example

The clock generator is an example of a circuit that functions correctly, yet cannot be simulated by *esim* [5]. *esim* is a switch-level simulator that accepts as input a transistor netlist from *magic*. The simulator state advances after values on all transistor nodes have converged to a steady state. Since the clock generator's normal function is to oscillate, *esim* cannot simulate the circuit; however, GES can extract the logical composition of the circuit to demonstrate that the correct components and the correct connections exist.

For this example, the clock generator will be extracted and VHDL code generated. Afterwards, the clock generator will be "corrupted" by introducing a design flaw into the circuit. A log of both sessions will be shown to demonstrate the error reporting feature of GES. The Prolog code used to extract the clock generator and produce VHDL code for the clock generator is shown in Sections 6 and 7, respectively.

Figure 9 is a layout diagram of the clock generator. The four largest pads are (starting from the top and going clockwise) OZ_PQ2, Vdd, OZ_PQ1, and GND. The three smaller pads are (starting from the top going clockwise) IZ_CAP1, IZ_GO, and IZ_CAP2. Both IZ_CAP1 and IZ_CAP2 are capacitively isolated from the rest of the circuit. Both did not appear in the transistor netlist using *mextra*; however, they did appear using *extract*. The input to GES used the output from *extract*, thus the capacitively isolated signals, IZ_CAP1 and IZ_CAP2, show up in the report as capacitive transistors.

The clock generator circuit consists of 116 p-type and n-type MOS transistors. The transistors in the transistor netlist, generated from the mask layout description using *extract*, form fully static CMOS components. The logical components were extracted from the transistor netlist using the level-2 and lower rules. The following is a listing of the log file.

```
% prolog

Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- compile(['new_trans_pos.pro']).
[compiling /usr/users/dukes/class/new_trans_pos.pro...]
[Undefined procedures will just fail ('fail' option)]
[new_trans_pos.pro compiled 14.200 sec 14,828 bytes]

yes
| ?- go.
finished with read.
Capacitor, ptrans(nIZ_CAP2,n3_140_8,n3_140_8,136,52): removed
```

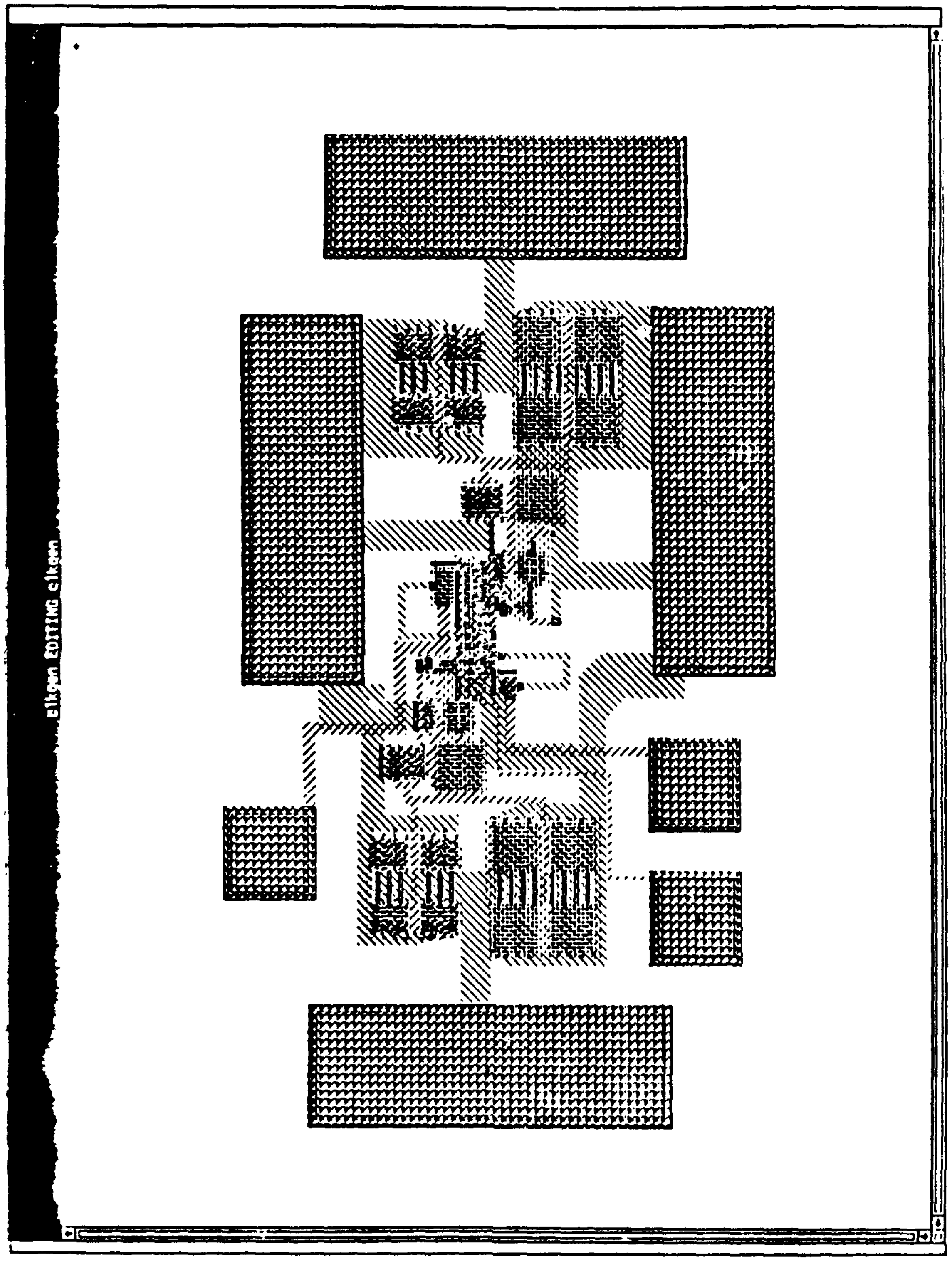

Figure 9. Layout Diagram of a Clock Generator Circuit.

```
Capacitor, ntrans(nIZ_CAP1,n3_12_115,n3_12_115,9,-42): removed
finished with find_error.
finished inverters.
finished tgates.
finished clk_inv.
finished nand.
finished nor.
finished find_more_errors.
%
```

The capacitively isolated signals **nIZ_CAP1** and **nIZ_CAP2** were found and removed. All other components in the circuit were successfully extracted using GES. Figure 10 is a listing of the components found using GES.

```
inv(n3_241_7,n3_673_317,-120,47,1).
inv(n3_72_106,n3_83_20,41,42,1).
inv(n3_83_20,n3_241_7,-37,36,1).
inv(n3_40_12,n3_66_21,35,6,1).
inv(n3_66_21,n3_12_115,48,8,1).
inv(n3_12_115,n3_140_8,70,7,1).
inv(n3_140_8,n3_44_29,99,8,1).
inv(n3_44_29,n3_226_12,113,9,1).
inv(n3_226_12,n3_330_10,182,11,1).
inv(n3_673_317,nOZ_PQ1,-334,-28,1).
inv(n3_238_103,n3_248_141,124,-62,1).
inv(n3_248_141,n3_324_69,169,-64,1).
inv(n3_324_69,n3_496_221,246,-62,1).
inv(n3_496_221,nOZ_PQ2,356,-64,1).
inv(nIZ_G0,n3_10_21,10,6,1).
nand([n3_10_21,n3_44_29],n3_40_12,18,-9,1).
nor([n3_238_103,n3_226_12],n3_72_106,141,8,1).
nor([n3_72_106,n3_330_10],n3_238_103,155,8,1).
```

Figure 10. Component Netlist of Clock Generator After Logic Extraction.

GES produced a listing of the components in the clock generator and their interconnections. From the extracted component netlist a circuit diagram, shown in Figure 11, was produced. The mask layout description of the clock generator was verified to have been generated correctly inasmuch as the components and their interconnections were concerned. The next example with the clock generator was performed to demonstrate how a faulty circuit can be highlighted.

The ring oscillator portion of the clock generator is shown in Figure 12. The ring oscillator is in the center of the layout diagram shown in Figure 9. Zooming further into the ring oscillator of the clock generator we see a portion of the circuit shown in Figure 12. A box was drawn in Figure 12 to show where the circuit in Figure 13 is located relative to the entire ring oscillator. The area where the circuit will be corrupted is shown within the box of Figure 13. To introduce a flaw into the circuit, the metal-1 line for GND and the metal-1 line output of an inverter are shorted together to demonstrate a possible human error during layout. Figure 14 shows the flawed circuit. Figure 15 shows a circuit diagram of the normal and abnormal portion of the affected circuit. To

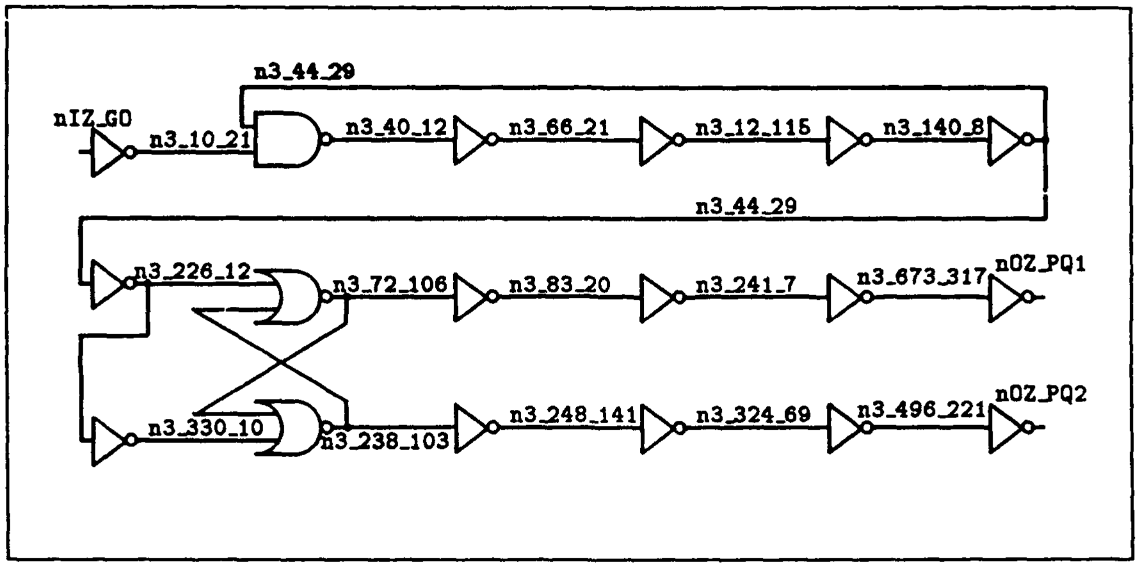

Figure 11. Circuit Diagram of the Extracted Clock Generator.

help make the example more interesting, a polysilicon line has also been severed. At this point there is a multiple fault in the circuit. The first fault demonstrates a stuck-at-0 fault, whereas the second demonstrates a floating fault on one portion of the inverter. The other half of the inverter has still been left to function. Figure 16 shows the layout of the entire corrupted clock generator.

For this relatively small circuit, the induced flaws are difficult to find by observation. Furthermore, in a circuit of 100K transistor size, these errors would be far more difficult to find under a probe station and may be undetectable while under test. However, GES can perform the tedious extraction and error identification. The following is a log of the Prolog session using GES on the corrupted clock generator circuit.

```
% prolog

Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- compile(['new_trans_pos.pro']).
[compiling /usr/users/dukes/class/new_trans_pos.pro...]
[Undefined procedures will just fail ('fail' option)]
[new_trans_pos.pro compiled 14.133 sec 14,828 bytes]

yes
| ?- go.
finished with read.
Bad trans, ptrans(n3_12_115,nvdd,ngnd,70,7): removed
Straight wire, ptrans(ngnd,nvdd,n3_44_29,99,8): removed
Capacitor, ptrans(n3_256_72,ngnd,ngnd,136,52): removed
```

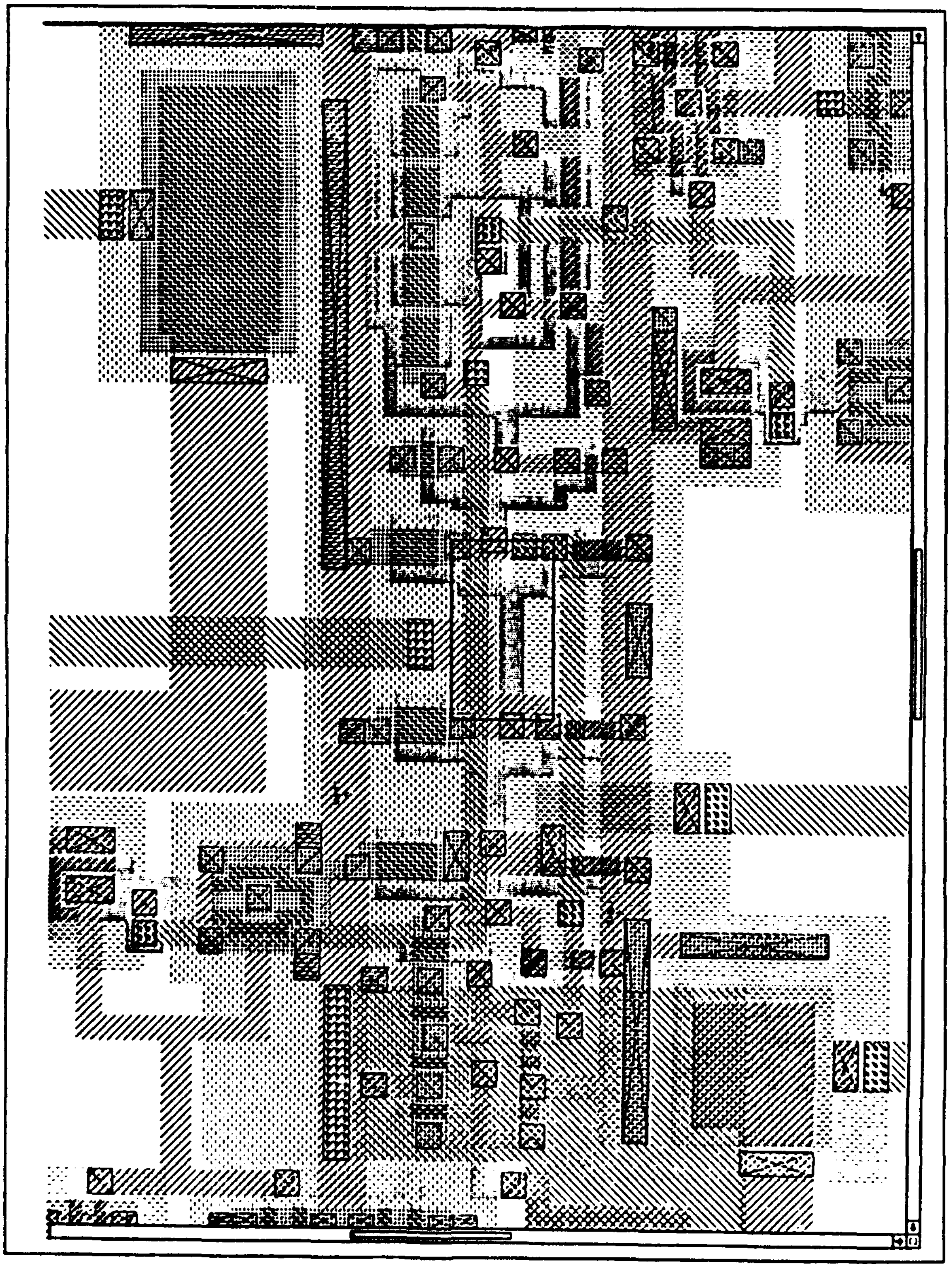
Figure 12. Ring Oscillator Portion of the Clock Generator.

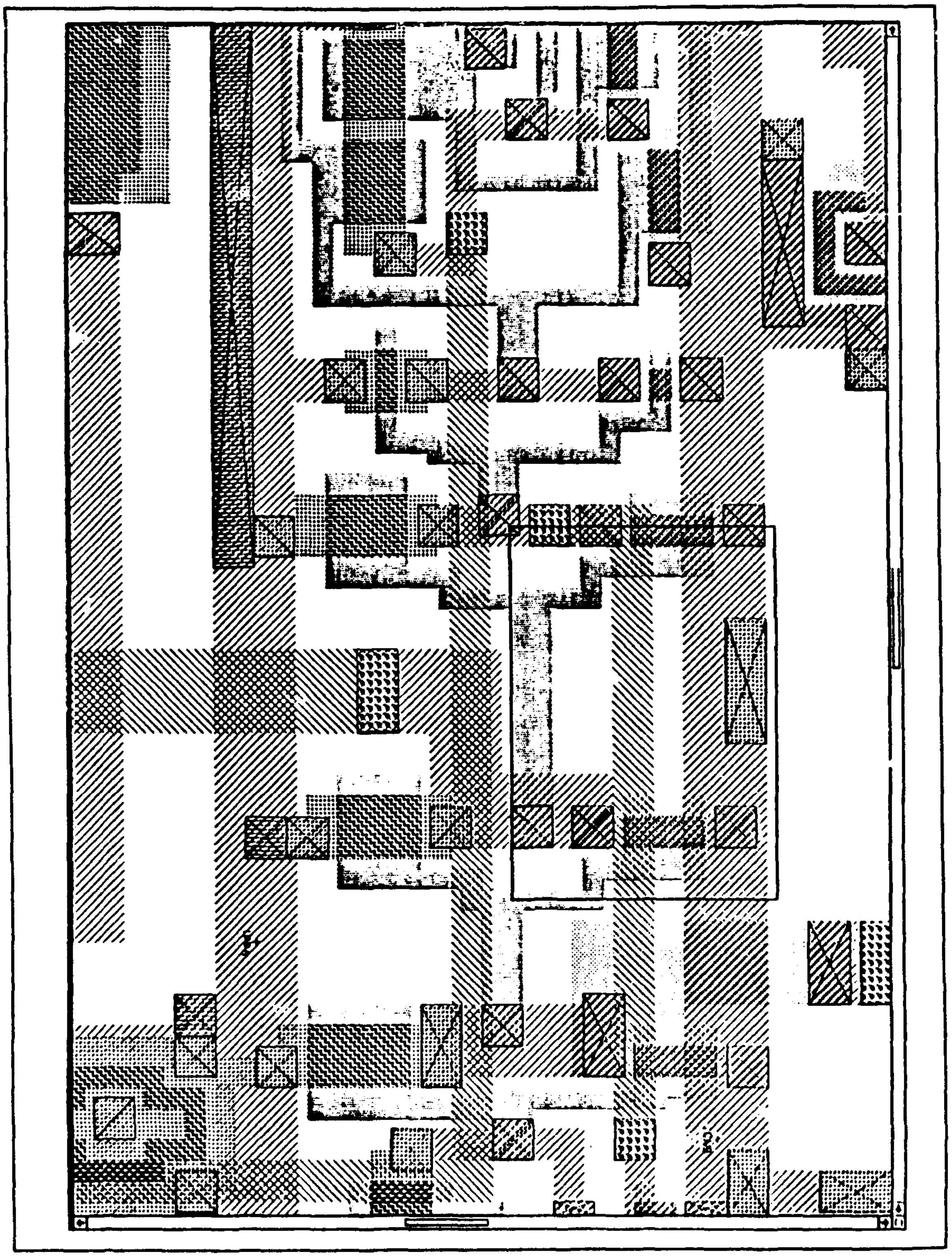

Figure 13. Zoom-in View of Ring Oscillator Circuit.

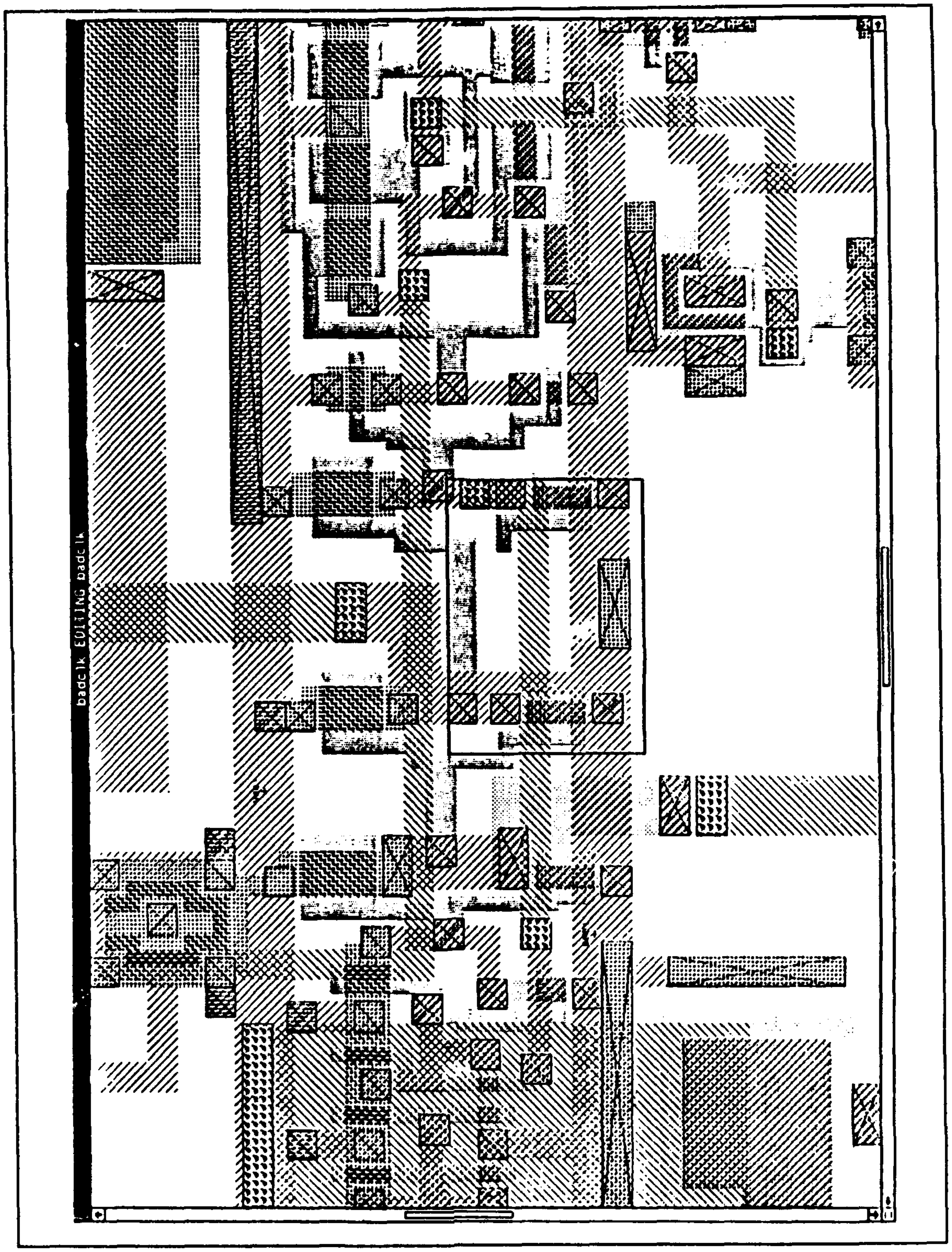

Figure 14. Induced Flaws in the Ring Oscillator.

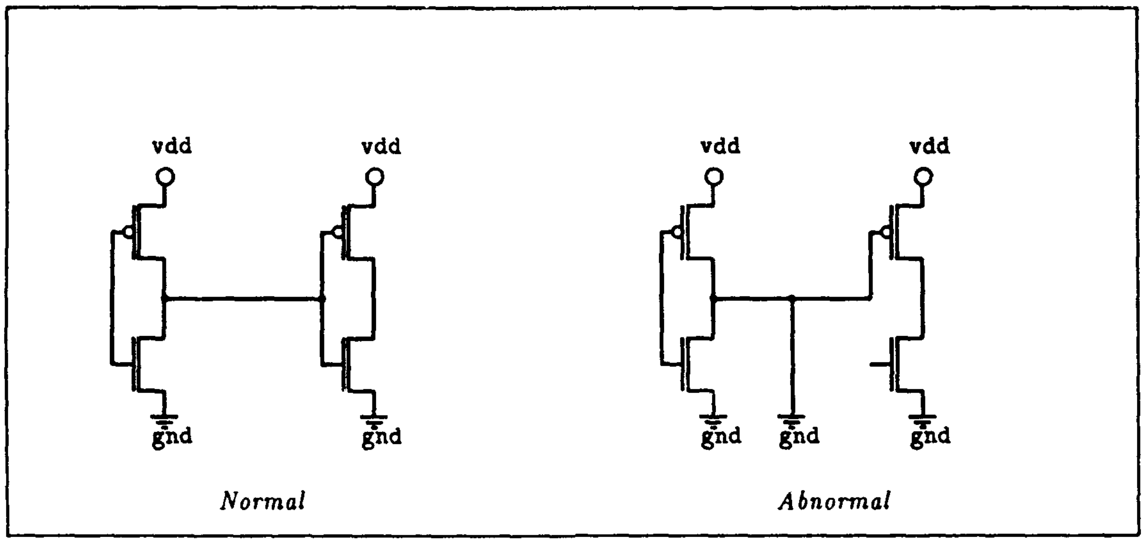

Figure 15. Circuit Diagram of Normal and Abnormal Circuit.

```
Capacitor, ntrans(n3_2_109,n3_12_115,n3_12_115,9,-42): removed
Capacitor, ntrans(n3_12_115,ngnd,ngnd,71,-21): removed
finished with find_error.
finished inverters.
finished tgates.
finished clk_inv.
finished nand.
finished nor.
finished find_more_errors.
```

Shown in Figure 17 is a component listing of the corrupted clock generator after using GES.

8.2 60,000 Transistor Design

This section outlines some performance results using Quintus Prolog to run GES. GES was used on a 60,000 transistor design of a multiplier section in a floating-point multiplier chip. The 60,000 transistor design performs multiplication on normalized mantissas of two floating-point numbers. The output of GES in this example proved to be very beneficial for the layout designers working on this design.

The **statistic/0** function of Quintus Prolog was used to report periodic timing statistics within the extraction process. These results appear in the table below.

The performance results were collected from a MicroVAX 3600 running Ultrix V3.1 using Quintus Prolog. 318 design errors were found in the VLSI design. From Definition E1, there were 30 type 1 errors, 36 type 3 errors, 10 type 4 errors, 10 type 5 errors, and 120 type 6 errors. There were also 30 capacitor-type transistors. From Definition E2, there were 82 type 4 errors. Some of the errors reported from in the original design are shown below.

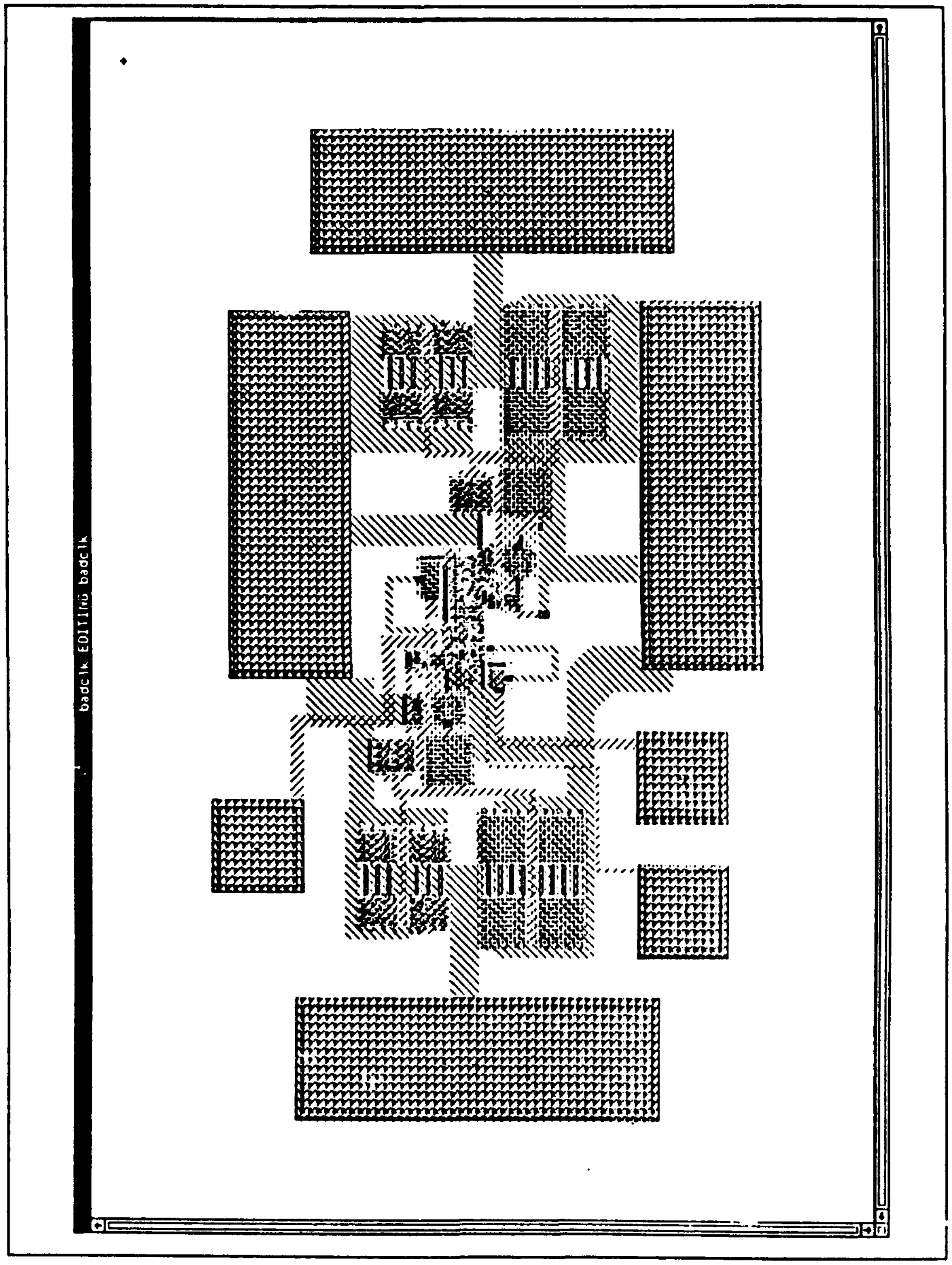

Figure 16. Corrupted Clock Generator Circuit.

```
n(n3_188_23,n3_44_29,ngnd,8,3,100,-22).
inv(n3_241_7,n3_673_317,-120,47,1).
inv(n3_72_106,n3_83_20,41,42,1).
inv(n3_83_20,n3_241_7,-37,36,1).
inv(n3_40_12,n3_66_21,35,6,1).
inv(n3_66_21,n3_12_115,48,8,1).
inv(n3_44_29,n3_226_12,113,9,1).
inv(n3_226_12,n3_330_10,182,11,1).
inv(n3_673_317,n3_669_313,-334,-28,1).
inv(n3_238_103,n3_248_141,124,-62,1).
inv(n3_248_141,n3_324_69,169,-64,1).
inv(n3_324_69,n3_496_221,246,-62,1).
inv(n3_496_221,n3_712_385,356,-64,1).
inv(n3_7_15,n3_10_21,10,6,1).
nand([n3_10_21,n3_44_29],n3_40_12,18,-9,1).
nor([n3_238_103,n3_226_12],n3_72_106,141,8,1).
nor([n3_72_106,n3_330_10],n3_238_103,155,8,1).
```

Figure 17. Component Listing of Faulted Clock Generator.

Table 1. Performance of GES on a 60,000 Transistor VLSI Design

Time to Perform	CPU Time (min:sec)
Read and Eliminate Duplicates	025:40
Find Definition E1 Errors	000:26
Find inv	002:56
Find tgates	683:15
Find invZ	121:32
Find nand	036:30
Find nor	025:21
Find Definition E2 Errors	000:05
Find dff	000:11
Write Component Netlist	007:24

```
Bad trans, ptrans(n14_038_0A_BAR,nvdd,ngnd,5335,3841): removed
Bad trans, ptrans(n14_038_1A_BAR,nvdd,ngnd,5335,3622): removed
Straight wire, ptrans(ngnd,nvdd,n13_043_0sumadder_13_916_23,457,4130): removed
Straight wire, ptrans(ngnd,nvdd,n13_044_0sumadder_13_916_23,727,4130): removed
Open connection, ptrans(nvdd,n14_039_03_302_425,n14_039_0COUT0,5461,3772):
removed
Open connection, ptrans(nvdd,n14_040_03_366_591,n14_040_0COUT0,5493,3553):
removed
Straight wire, ntrans(nvdd,ngnd,n14_038_03_214_111,5409,3817): removed
Straight wire, ntrans(nvdd,ngnd,n14_038_13_214_111,5409,3598): removed
Open connection, ntrans(ngnd,ngnd,n13_043_0sumadder_13_916_23,457,4115):
removed
Open connection, ntrans(ngnd,n13_043_0sumadder_13_868_27,
n13_043_0sumadder_13_820_27,441,4113): removed
Capacitor, ntrans(n14_038_0A_BAR,ngnd,ngnd,5335,3821): removed
Capacitor, ntrans(n14_038_1A_BAR,ngnd,ngnd,5335,3602): removed
Screwy tgate, tgate(n19_031_5XOR,n19_031_5XOR,n19_031_5COUT1,
n19_030_16COUT1,5018,-73,1): removed
Screwy tgate, tgate(n19_032_10XOR,n19_032_10XOR,n19_032_10COUT1,
n19_030_15COUT1,5034,73,1): removed
```

From the performance measurement of GES in Table 1 we can make a few observations. First, I/O for the example had moderate impact on the execution time of GES. Secondly, looking for errors in the design can be done very quickly. This factor is indeed desirable, since a designer would want to interact with GES in an attempt to find errors that may exist within a layout design. Though the timing statistics do not represent many level-N components, extraction for other types of components (e.g., half-adders, adders, adder-arrays, registers) usually progresses quickly. The vast amount of extraction time spent is usually with the level-1 and level-2 rules.

IX. Conclusion

GES has been used to verify several designs. In the case of the clock generator, a design was verified that could not be simulated using *esim*. Through the logic extraction process, GES verifies that the correct components and interconnections exist within a VLSI design. GES is very helpful at finding logical design errors when the correct components and interconnections do not exist. The error reporting feature provides immediate feedback that is hard to get from the Berkeley CAD tools. Furthermore, GES provides logic extraction in a reasonable amount of CPU time and memory for VLSI-sized chip descriptions. The resulting VHDL and HOL code provides a description of what actually exists in the VLSI layout.

Though this paper has focused on CMOS design styles, other types of design styles may be incorporated into GES. Logic extraction using Prolog allows for easy rule formation, allowing the system to be applicable over a number of technologies used in VLSI layout. Applications involving laser programmable designs may also benefit from GES. Cuts and splices may be performed in a layout, using GES to check the new design for errors and to generate VHDL for the actual component.

GES is easily modified. Following the extraction rule format provides the user with the flexibility to generate new extraction rules that describe and check newly-created cells. The declarative nature of Prolog allows the user to concentrate on how a component is constructed. In this manner, GES easily accommodates hierarchical design and performs best when hierarchy is used.

X. References

1. California, University of, at Berkeley. Berkeley Distribution of Design Tools. Computer Science Division, EECS Department, University of California at Berkeley; 1986.
2. Clocksin, W. F. and C. S. Mellish. *Programming in Prolog.* New York: Springer-Verlag; 1987.
3. Bratko, Ivan. *Prolog Programming for Artificial Intelligence.* Reading: Addison-Wesley Publishing Company, Inc.; 1986.
4. IEEE, Computer Society Standards Committee, "IEEE Standard VHDL Language Reference Manual," *ANSI/IEEE Std 1076-1987*, New York: IEEE Press; 1987.
5. Terman, Chris. "Esim," Berkeley Distribution of Design Tools. Computer Science Division, EECS Department, University of California at Berkeley; 1986.
6. Weste, N. H. and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Reading: Addison-Wesley Publishing Company; 1985.
7. Sterling, Leon and Ehud Shapiro. *The Art of Prolog.* Cambridge: The MIT Press; 1986.
8. Gordon, Michael. "HOL: A Proof Generating System for Higher-Order Logic," *VLSI Specification, Verification, and Synthesis*, Boston: Kluwer Academic Publishers; pp. 73-128, 1988.
9. Gordon, Michael. *The HOL System Tutorial.* Cambridge Research Center of SRI International under a grant from DSTO Australia, 8 December 1989.
10. Cousineau, G., G. Huet and L. Paulson. *The ML Handbook.* INRIA; 1986.
11. de Geus, Aart J. "Logic Synthesis Speeds ASIC Design," *IEEE Spectrum*, vol. 26, no. 8, pp. 27-31, 1989.

Appendix A. *Definitions*

A.1 Definition of Behavioral and Structural Specification

In this section, a definition will be given for behavioral specification and structural specification. For the purposes of the presentation, combinational logic will be used. Afterwards, an example of each in VHDL will be offered to help distinguish the two forms of specification.

A behavioral specification is a relation that may be described algorithmically. A behavioral specification also describes how a specified system or component is expected to react to a given set of input stimuli. There is usually nothing or very little provided in the behavioral specification as to the internal physical makeup and interconnections of the specified system or component. The behavioral specification may be represented abstractly as in Figure 18. The boundary of the specified system or component is well defined. By well defined we mean that all inputs and outputs are identified at the boundary of the specified device or component.

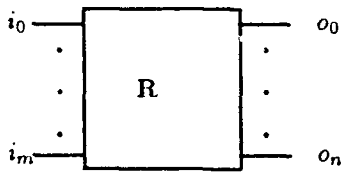

Figure 18. Abstract View of a Behavioral Specification.

From Figure 18, the inputs to the specified device or component are represented as I where $I = (i_0, ..., i_m)$ and $0 \leq m$. The outputs from the specified device or component are represented as O where $O = (o_0, ..., o_n)$ and $0 \leq n$. The behavior for the specified device or component is represented by a relation $\mathbf{R}$ where $\mathbf{R} \subset I \times O$.

A structural specification for a specified device or component provides a description of the internal physical makeup and interconnections. The following criteria are used to determine a structural specification.

1. A structural specification is not a behavioral specification.
2. A structural specification may be constructed from one or more interconnected behavioral specifications.
3. A structural specification may be constructed from one or more interconnected structural specifications.
4. A structural specification may be constructed from one or more interconnected behavioral specifications and structural specifications.

A behavioral specification may be viewed as a procedural method for portraying a specified device or component. Alternatively, a structural specification may be viewed as a declarative method for portraying a specified device or component.

The following is a VHDL model of a behavioral specification.

```
entity adder is
  port(i0,i1,i2: in  bit;
       o0,o1    : out bit);
  end adder;

architecture behave of adder is

  function bv (input : bit) return integer is
    begin
    If(input = '1') then return(1);
    else return(0); end if;
    end;

  function bv_inv (input : integer) return bit is
    begin
    If(input = 0) then return('0');
    else return('1'); end if;
    end;

  begin

  process

    begin
    wait on i0,i1,i2;
    if ((bv(i0)+bv(i1)+bv(i2)) < 2) then
      o0 <= bv_inv(bv(i0)+bv(i1)+bv(i2));
    else
      o0 <= bv_inv(bv(i0)+bv(i1)+bv(i2)-2);
    end if;
    if ((bv(i0)+bv(i1)+bv(i2)) < 2) then
      o1 <= bv_inv(0);
    else
      o1 <= bv_inv(1);
      end if;
    end process;
  end behave;
```

From the VHDL description the inputs and output are enumerated as i0, i1, i2 for the inputs and o0,o1 for the outputs. The assigned values for the outputs are determined algorithmically from the inputs. There is nothing in the VHDL description that indicates the physical construction of the specified device or component.

The following VHDL description is a structural specification of the same device.

```
entity adder is
  port(i0,i1,i2: in  bit;
       o0,o1    : out bit);
  end adder;
```

```
architecture structure of adder is

  signal p,q,r : bit;

  component half_add
    port (a,b : in bit;
          s,cbar : out bit);
    end component;

  begin

  o1 <= q nand r;

  ha1 : half_add port map
    ( a => i0,
      b => i1,
      s => p,
      cbar => q);

  ha2 : half_add port map
    ( a => p,
      b => i2,
      s => o0,
      cbar => r);

  end structure;
```

A.2 Definitions for Other Terms

Logic Extraction creation of a higher level digital component by recognizing a collection of one more lower level components and interconnections that make up the higher level digital component.

Hardware description language (HDL) "a language used to describe a circuit's behavior or structure." [11]

Logic synthesis "creation of a gate-level netlist from a register-transfer level description." [11]

Mapping "the process of formulating a design in terms of the cells available in a given parts library." [11]

Netlist "a circuit design description in terms of structural elements and their interconnections." [11] The term component netlist is synonymous with netlist.

Transistor Netlist a netlist consisting only of transistors.

Appendix B. *LEX Program for Mapping Labels to Upper Case*

Listed in this appendix is a lex[1] routine used to map lower and upper case letters on node labels to strictly upper case letters for node labels.

```
%START AA BB
     int i;
%%
^94[ ]Vdd![ ]([0-9]|\-)[^\n]*/\n        ECHO;
^94[ ]GND![ ]([0-9]|\-)[^\n]*/\n        ECHO;
^94[ ]                   {
                         ECHO;
                         BEGIN AA;
                         }
<AA>[^ ]+                {
                         for (i=0; i<yyleng; i++)
                            if (yytext[i] >= 'a' && yytext[i] <= 'z')
                              printf("%c",(yytext[i]+'A'-'a'));
                            else
                              printf("%c",yytext[i]);
                         BEGIN BB;
                         }
<BB>[^\n]*/\n            {
                         ECHO;
                         BEGIN 0;
                         }
```

An executable binary is generated from the lex routine in the following manner. For the example, assume the name of the lex routine is `upper.lex` and the UNIX system prompt is `csh>`.

```
csh> lex upper.lex
csh> cc lex.yy.c -ll
```

The executable binary produced is called `a.out`.

[1]UNIX lexical analysis program generator

Appendix C. *C Program for Converting Transistor Netlists*

The following C program is used to convert a transistor netlist produced by *mextra* [1] into a transistor netlist in Prolog clause form.

```
#include <stdio.h>

#define max_buf 128

char buffer[max_buf];
char tempbuf[max_buf];

int time_count,
    iteration,
    count,count2;



FILE *fd,*od;



main()
{
fd=fopen("new.sim","r");
od=fopen("good.pro","w");
for(count=0;count<max_buf;count++)
    {
    tempbuf[count]=0;
    }
fgets(buffer,max_buf,fd);
while(fgets(buffer,max_buf,fd) != NULL)
    {
    if(buffer[0]=='e')
        {
        count=3;
        count2=2;
        iteration=0;
 tempbuf[0]='n';
 tempbuf[1]='(';
        tempbuf[2]='n';
        while((buffer[count]!=0)&(iteration!=3))
            {
            if ((buffer[count2]=='V')&(buffer[count2+1]=='d')&
              (buffer[count2+2]=='d'))
                {
                --count;
                tempbuf[count++]='v';
                tempbuf[count++]='d';
                tempbuf[count]='d';
                count2=count2+2;
                }
            else if ((buffer[count2]=='G')&(buffer[count2+1]=='N')&
                   (buffer[count2+2]=='D'))
                {
                --count;
                tempbuf[count++]='g';
                tempbuf[count++]='n';
                tempbuf[count]='d';
                count2=count2+2;
                }
            else if(buffer[count2]==' ')
                {
                tempbuf[count++]=',';
                tempbuf[count]='n';
                iteration++;
                }
            else if(buffer[count2]=='#')
                {
                --count;
                }
            else
                {
                tempbuf[count]=buffer[count2];
                }
            count++;
            count2++;
            }
            count=count-2;
    tempbuf[count++]=')';
            tempbuf[count++]='.';
            tempbuf[count++]=10;
            tempbuf[count]=0;
        for(count=0;count<max_buf;count++)
            {
            buffer[count]=tempbuf[count];
            }
        fprintf(od,"%s",buffer);
        for(count=0;count<max_buf;count++)
            {
            tempbuf[count]=0;
            }
        }
    else if(buffer[0]=='p')
        {
        count=3;
        count2=2;
        iteration=0;
tempbuf[0]='p';
tempbuf[1]='(';
        tempbuf[2]='n';
        while((buffer[count]!=0)&(iteration!=3))
            {
```

```
          if ((buffer[count2]=='V')&(buffer[count2+1]=='d')&
            (buffer[count2+2]=='d'))
              {
              --count;
              tempbuf[count++]='v';
              tempbuf[count++]='d';
              tempbuf[count]='d';
              count2=count2+2;
              }
          else if ((buffer[count2]=='G')&(buffer[count2+1]=='N')&
                (buffer[count2+2]=='D'))
              {
              --count;
              tempbuf[count++]='g';
              tempbuf[count++]='n';
              tempbuf[count]='d';
              count2=count2+2;
              }
          else if(buffer[count2]==' ')
              {
              tempbuf[count++]=',';
              tempbuf[count]='n';
              iteration++;
              }
          else if(buffer[count2]=='#')
              {
              --count;
              }
          else
              {
              tempbuf[count]=buffer[count2];
              }
          count++;
          count2++;
          }
          count=count-2;
tempbuf[count++]=')';
          tempbuf[count++]='.';
          tempbuf[count++]=10;
          tempbuf[count]=0;
      for(count=0;count<max_buf;count++)
          {
          buffer[count]=tempbuf[count];
          }
      fprintf(od,"%s",buffer);
      for(count=0;count<max_buf;count++)
          {
          tempbuf[count]=0;
          }
      }
}
```

```
fclose(fd);
fclose(od);
}
```

Appendix D. *Using GES at AFIT*

This appendix is meant for the first-time user of GES. Therefore, the context of what follows is a walk-through session of GES with a VLSI layout design and a corrupted version of the same VLSI design. The first section shows logic extraction and error identification using GES. The second and third sections demonstrate the generation of VHDL and HOL with GES, respectively. The fourth section outlines a GES session with a corrupted VLSI design. The final section contains an exercise. The user is encouraged to examine all designs using *magic*. Furthermore, the design errors identified in the fourth section should be located in *magic* in order to gain familiarity with finding errors reported by GES.

D.1 Performing Logic Extraction and Error Identification

Outlined in this section is an explanation and example of how to use GES. This section is helpful for the individual who has little concern of how GES works and wishes to use the pretailored GES system for a custom VLSI circuit designed at the Air Force Institute of Technology (AFIT).

Prior to beginning this example, the following had been added as the last line in the `.login` file.

```
set path=($path ~cad/bin ~ges/bin)
```

The **set path** line makes the *magic*, *ext2sim*, *sim2pro*, and *ges_base_cmos* routines visible to the login shell. The example begins from a system other than CUB, demonstrating how to access CUB. The system prompt is denoted by `[1]cub` .

```
% telnet cub
Trying...
Connected to cub.afit.af.mil.
Escape character is '^]'.
cub login: mdukes
Password:
Last login: Fri Aug  3 13:19:31 from galaxy
Ultrix-32 V3.1 (Rev. 9) System #6: Wed Jul 25 09:46:06 EDT 1990

        The system that brought Higher Order Logic
                       to AFIT

  Maybe Computer Science should be in the College of Theology.
                -- R. S. Barton

Fri Aug  3 13:45:54 EDT 1990
[1]cub
```

Two *magic* examples have been provided under `~ges/example`. To gain access to these examples for the rest of this section, copy them as shown below.

```
[1]cub mkdir GES
[2]cub cd GES
[3]cub cp ~ges/example/* .
[4]cub ls
badclk.mag  clkgen.mag  component  prob1.mag
[5]cub
```

magic is started using the NULL device. The layout file for this example is called **clkgen.mag**.

```
[5]cub magic -d NULL clkgen

Magic - Version 4.10 - Last updated 11/18/88 at 16:11:26

Using technology "scmos".
Using NULL graphics device.
clkgen: 500 rects
clkgen: 1000 rects
clkgen: 1500 rects
clkgen: 2000 rects
clkgen: 2500 rects
clkgen: 3000 rects
clkgen: 3500 rects
clkgen: 4000 rects
clkgen: 4500 rects
clkgen: 5000 rects
clkgen: 5500 rects
clkgen: 6000 rects
clkgen: 6500 rects
clkgen: 7000 rects
clkgen: 7500 rects
clkgen: 8000 rects
clkgen: 8500 rects
>
```

Once the *magic* file is loaded in, the layout is extracted using *magic*'s **:extract** command.

```
:extract
Extracting clkgen into clkgen.ext:
>
```

Once the layout is extracted, the *magic* session is terminted by using the tt :quit command (may be abbreviated to **:q**). Afterwards, *ext2sim* is executed with the switches **-R -C -A** to eliminate resistors, capacitors, and aliases in the final file. This will also reduce the execution time of *ext2sim*.

```
:q
[6]cub ext2sim -R -C -A clkgen
```

In order to convert the resulting **clkgen.sim** from *ext2sim* to a format for use by GES, the file must be moved to **new.sim**. Then a routine called *sim2pro* is executed to convert from an *esim* formate to a *Prolog* format.

```
[7]cub mv clkgen.sim new.sim
[8]cub sim2pro
```

To execute the GES baseline system, enter the command *ges_base_cmos* at the system prompt. Once GES has loaded the necessary libraries, the user is prompted with | ?- . At the prompt, type **ges** followed by a period.

```
[9]cub ges_base_cmos

yes
| ?- ges.
finished with read.
Capacitor, ptrans(nIZ_CAP2,n3_140_8,n3_140_8,136,52): removed
Capacitor, ntrans(nIZ_CAP1,n3_12_115,n3_12_115,9,-42): removed
finished with find_error.
finished inverters.
finished tgates.
finished invZ.
finished nand.
finished nor.
finished find_more_errors.
finished dff.
[10]cub
```

As each component type is extracted and errors are found, a report is made. Once GES has finished, the system prompt reappears. There are two resulting files, **outcomp** containing the extracted components and **outtrans** containing the remaining transistors. The file **outcomp** is shown below.

```
inv(n3_241_7,n3_673_317,-120,47,1).
inv(n3_72_106,n3_83_20,41,42,1).
inv(n3_83_20,n3_241_7,-37,36,1).
inv(n3_40_12,n3_66_21,35,6,1).
inv(n3_66_21,n3_12_115,48,8,1).
inv(n3_12_115,n3_140_8,70,7,1).
inv(n3_140_8,n3_44_29,99,8,1).
inv(n3_44_29,n3_226_12,113,9,1).
inv(n3_226_12,n3_330_10,182,11,1).
inv(n3_673_317,nOZ_PQ1,-334,-28,1).
inv(n3_238_103,n3_248_141,124,-62,1).
inv(n3_248_141,n3_324_69,169,-64,1).
inv(n3_324_69,n3_496_221,246,-62,1).
inv(n3_496_221,nOZ_PQ2,356,-64,1).
inv(nIZ_GO,n3_10_21,10,6,1).
nand([n3_10_21,n3_44_29],n3_40_12,18,-9,1).
nor([n3_238_103,n3_226_12],n3_72_106,141,8,1).
nor([n3_72_106,n3_330_10],n3_238_103,155,8,1).
```

The file **outtrans** contained no transistors.

D.2 Generating VHDL

In order to generate VHDL from an extracted component netlist, three files must exist. These files are **outcomp**, **outtrans**, and **component**. The first two files are automatically generated after successful completion of GES. The third file, **component**, must be generated by the user. The content of the **component** file for this example is shown below.

```
component(clkgen,[nIZ_GO,nIZ_CAP1,nIZ_CAP2],[nOZ_PQ1,nOZ_PQ2],[],[],[]).
```

The **component** file contains the name of the component being extracted (in this case, clkgen). It also contains five lists, a list of signals of mode in, a list of signals of mode out, a list of signals of mode inout, a list of signals of mode buffer, and a list of signals of mode linkage[1].

To generate VHDL, type the command *pro2vhdl*. Once the | ?- appears, type **pro2vhdl.** to start the VHDL generation. The VHDL output file is placed in **outfile.vhd**.

```
[10]cub pro2vhdl

yes
| ?- pro2vhdl.
Finished with read
[11]cub
```

D.3 Generating HOL

As with the previous section concerning VHDL generation, HOL generation requires the same three files. The same **component** file will be used from the previous section. Enter *pro2hol* at the system prompt. Once the | ?- prompt appears type the command **pro2hol.** again. The HOL output will be placed in a file called **outfile.ml** which may be read into HOL using the **loadt** HOL command.

[1]the modes referred to in this context are the modes explained in §4.3.3 of the *VHDL Language Reference Manual* [3]

```
[12]cub pro2hol

yes
| ?- pro2hol.
Finished with read
[13]cub
```

D.4 Finding Design Errors

This section uses the corrupted version of the clock generator to demonstrate how GES identifies design errors. The corrupted version of the clock generator should have been copied into the working directory as directed at the beginning of this appendix.

The *magic*, *ext2sim*, and *sim2pro* commands are entered as they were earlier.

```
[14]cub magic -d NULL badclk

Magic - Version 4.10 - Last updated 11/18/88 at 16:11:26

Using technology "scmos".
Using NULL graphics device.

Warning -- cell badclk not writable
badclk: 500 rects
badclk: 1000 rects
badclk: 1500 rects
badclk: 2000 rects
badclk: 2500 rects
badclk: 3000 rects
badclk: 3500 rects
badclk: 4000 rects
badclk: 4500 rects
badclk: 5000 rects
badclk: 5500 rects
badclk: 6000 rects
badclk: 6500 rects
badclk: 7000 rects
badclk: 7500 rects
badclk: 8000 rects
badclk: 8500 rects
:extract
Extracting badclk into badclk.ext:
badclk: 3 warnings
Total of 3 warnings.
:q
[15]cub ext2sim -R -C -A badclk
[16]cub cp badclk.sim new.sim
[17]cub sim2pro
[18]cub
```

GES is executed using *ges_base_cmos*.

```
[18]cub ges_base_cmos

yes
| ?- ges.
finished with read.
Bad trans, ptrans(n3_12_115,nvdd,ngnd,70,7): removed
Straight wire, ptrans(ngnd,nvdd,n3_44_29,99,8): removed
Capacitor, ptrans(nIZ_CAP2,ngnd,ngnd,136,52): removed
Capacitor, ntrans(nIZ_CAP1,n3_12_115,n3_12_115,9,-42): removed
Capacitor, ntrans(n3_12_115,ngnd,ngnd,71,-21): removed
finished with find_error.
finished inverters.
finished tgates.
finished invZ.
finished nand.
finished nor.
finished find_more_errors.
finished dff.
[19]cub
```

Several transistors have been identified as errors. The contents of the resulting outcomp are shown below.

```
inv(n3_241_7,n3_673_317,-120,47,1).
inv(n3_72_106,n3_83_20,41,42,1).
inv(n3_83_20,n3_241_7,-37,36,1).
inv(n3_40_12,n3_66_21,35,6,1).
inv(n3_66_21,n3_12_115,48,8,1).
inv(n3_44_29,n3_226_12,113,9,1).
inv(n3_226_12,n3_330_10,182,11,1).
inv(n3_673_317,nOZ_PQ1,-334,-28,1).
inv(n3_238_103,n3_248_141,124,-62,1).
inv(n3_248_141,n3_324_69,169,-64,1).
inv(n3_324_69,n3_496_221,246,-62,1).
inv(n3_496_221,nOZ_PQ2,356,-64,1).
inv(nIZ_GO,n3_10_21, ),6,1).
nand([n3_10_21,n3_44_29],n3_40_12,18,-9,1).
nor([n3_238_103,n3_226_12],n3_72_106,141,8,1).
nor([n3_72_106,n3_330_10],n3_238_103,155,8,1).
```

The contents of the resulting outtrans are shown below.

```
n(n3_188_23,n3_44_29 ngnd,8,3,100,-22).
```

Figure 19 shows a circuit diagram for the normal and abnormal circuits.

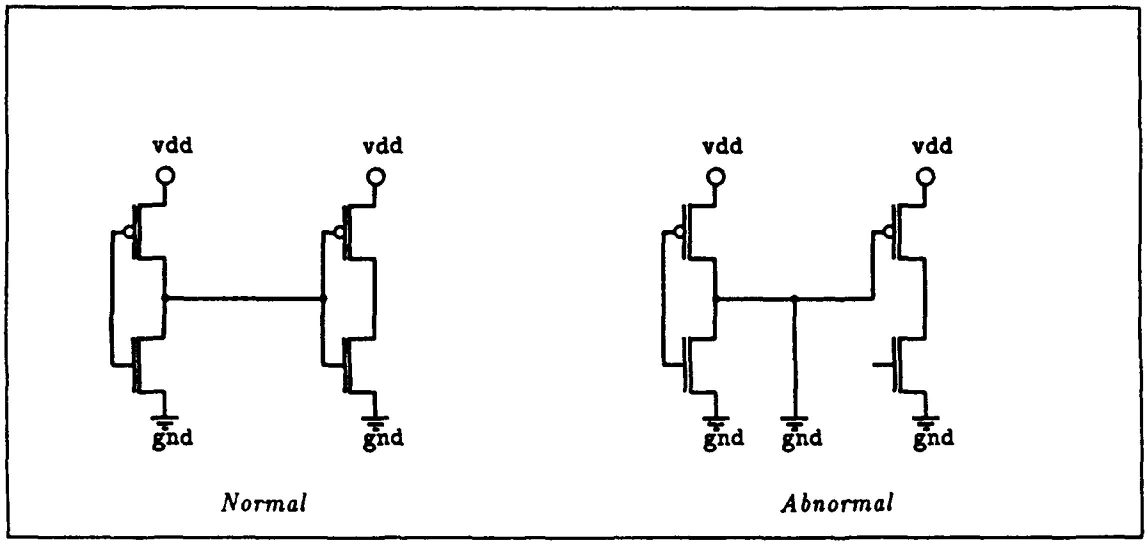

Figure 19. Circuit Diagram from **clkgen.mag** and **badclk.mag**.

D.5 Exercises

1. In **ges/example** is a *magic* file called **prob1.mag**. The file is a copy of the clock generator example except that a design error exists within the layout. Copy the file into your directory. Use GES to find the error. What do you notice about the **outcomp** file in both the normal clock generator and the corrupted clock generator? Is the **outcomp** file in both the normal clock generator and the corrupted clock generator the same or different? Check the **outtrans** file for both, also. Find and explain the error. Produce a plot of the errant area. (HINT: remember that GES removes duplicates prior to extraction)